HOCHSCHULE
MITTWEIDA
UNIVERSITY OF
APPLIED SCIENCES

DIPLOMARBEIT

Herr Ing.
Johannes Troppacher

Echtzeitspektralanalyse auf Basis der Fourier- und Wavelet-Transformation implementiert als VST-Plugin

2011

Fakultät **Elektro- und Informationstechnik**

DIPLOMARBEIT

Echtzeitspektralanalyse auf Basis der Fourier- und Wavelet-Transformation implementiert als VST-Plugin

Autor:
Johannes Troppacher

Studiengang:
Informationstechnik

Seminargruppe:
KI09wIA

Erstprüfer:
Prof. Dr.-Ing. Alexander Lampe

Zweitprüfer:
Dipl. Ing. Norbert Göbel

Mittweida, 07 2011

Bibliografische Angaben

Troppacher, Johannes: Echtzeitspektralanalyse auf Basis der Fourier- und Wavelet-Transformation implementiert als VST-Plugin, 107 Seiten, 31 Abbildungen, Hochschule Mittweida (FH), Fakultät Elektro- und Informationstechnik

Diplomarbeit, 2011

Johannes Troppacher

Referat

Im Rahmen dieser Diplomarbeit wird die Entwicklung eines VST-Spektralanalyse-Plugins beschrieben, das neben der Fourier- auch die Wavelet-Transformation zur Bestimmung der spektralen Bestandteile eines Audiosignals nutzt. Den Ausgangspunkt stellen die theoretischen Grundlagen auf dem Gebiet der Spektralanalyse und der Psychoakustik dar. Diese Grundlagen fließen in die praktische Umsetzung der Software ein, welche ausgehend von der Aufgabenstellung, über das Konzept bis hin zur Implementierung detailliert beschrieben wird.

I. Inhaltsverzeichnis

II. Abbildungsverzeichnis

III. Tabellenverzeichnis

IV. Abkürzungsverzeichnis

API ..... Application Programming Interface, Seite 56

AWT ..... Abstract Window Toolkit, Seite 56

CPU ..... CentralProcessingUnit, Seite 74

CWT ..... Continuous Wavelet Transformation, Seite 19

DFT ..... Diskrete Fourier Transformation, Seite 11

DLL ..... Dynamic Link Library, Seite 53

DWMT ..... Discrete Wavelet Multi-Tone, Seite 37

DWT ..... Diskrete Wavelet Transformation, Seite 25

FFT ..... Fast Fourier Transformation, Seite 13

FIR ..... Finite Impulse Response, Seite 31

FWT ..... Fast Wavelet Transformation, Seite 30

GNU ..... General Public Licence, Seite 56

GUI ..... Graphical User Interface, Seite 56

IDFT ..... Inverse Diskrete Fourier Transformation, Seite 12

IFWT ..... Inverse Fast Wavelet Transformation, Seite 30

MIT ..... Massachusetts Institute of Technology, Seite 58

MSA ..... Multi Skalen Analyse, Seite 27

OFDM ..... Orthogonal Frequency-Division Multiplexing, Seite 37

PR ..... Perfect Reconstruction, Seite 32

QMF ..... Quadrature Mirror Filters, Seite 36

SDK ..... Software Development Kit, Seite 53

SIMD ..... Single Instruction Multiple Data, Seite 14

SSE ..... Streaming SIMD Extensions, Seite 14

STDFT ..... Short Time Diskrete Fourier Transformation, Seite 15

VST^{TM} ..... Virtual Studio Technology^{TM}, Seite 53

WPT ..... Wavelet Packet Transform, Seite 32

1 Einleitung

Die digitale Audiosignalverarbeitung hat sich von der einst preisgünstigen Alternative zur Analogtechnik zu einer nicht mehr wegzudenkenden Basistechnologie entwickelt. Neben der Nachbildung analoger Schaltungen und Signalwege, welche bei manchen *Puristen* noch umstritten ist, ermöglicht sie auch Anwendungen, die auf keinem anderen Wege realisierbar wären.

Die Spektralanalyse stellt ebenfalls einen jener Bereiche dar, welche ohne digitale Signalverarbeitung undenkbar wären. In der Praxis hat sich dabei die Fourier-Transformation als Verfahren zur Ermittlung der Signalbestandteile durchgesetzt. Diese wird von zahlreicher Soft- und Hardware unterschiedlichster Plattformen unterstützt und gilt daher als besonders leistungsfähig und einfach umsetzbar.

Die Wavelet-Transformation findet dagegen in der Spektralanalyse, insbesondere im Audiobereich, noch kaum Anwendung, obwohl diese auf anderen Gebieten, wie z.B. der Datenkompression, längst nicht mehr nur als Geheimtipp gehandelt wird. Neben der Einführung eines Zeitbezugs, welcher zu einer, verglichen mit der Fourier-Transformation, eleganteren Lösung des Fensterungsproblems beiträgt, ermöglicht die Wavelet-Transformation und deren Varianten auch eine flexiblere Gestaltung des Frequenz/Zeit-Rasters.

Abgesehen von der Umsetzung digitaler Signalverarbeitung in Form von Hardware, z.B. als integrierter Schaltkreis, gewinnt die Umsetzung in Form von Software, dank immer leistungsfähigerer Standardrechner, im Audiobereich immer mehr an Bedeutung. Ein deutlicher Trend zeichnet sich dabei in der Entwicklung von Plugins ab, welche sich flexibel in unterschiedlichste Audioprogramme einbinden lassen. VST stellt hierbei einen der ältesten und gleichzeitig verbreitetsten Plugin-Standards dar.

Für die Spektralanalyse existieren bereits zahlreiche, zum Teil auch kostenfreie VST-Plugins. Keines dieser Plugins bietet allerdings die Möglichkeit, zwischen Fourier- und Wavelet-Transformation, sowie deren Varianten, zu wählen. Im Zuge dieser Arbeit wird die Entwicklung eines solchen VST-Plugins beschrieben.

Zu Beginn werden die Grundlagen der Spektralanalyse und der Psychoakustik behandelt. Anschließend werden derzeit am Markt befindliche Plugins verglichen und die Anforderungen an das zu entwickelnde Plugin definiert. Daraus wird dann das Konzept erarbeitet, welches die Evaluierung verschiedener Bibliotheken und Frameworks, sowie

das Software-Design und die Software-Architektur umfasst. Schließlich wird im Zuge der Implementierung konkret auf drei Problemlösungen eingegangen, welche anhand von Code-Ausschnitten detailliert analysiert werden. Abschließend wird das Ergebnis vorgestellt und mit den am Markt befindlichen Mitbewerbern verglichen. Die daraus resultierenden Erkenntnisse fließen in den Ausblick ein, in dem Erweiterungsmöglichkeiten diskutiert werden.

2 Grundlagen der Spektralanalyse

2.1 Einleitung

Viele in der Natur vorkommenden Ereignisse, wie Licht oder Schall, scheinen bei oberflächlicher Beobachtung rein zufällig. Erst bei genauerer Untersuchung lassen sich Muster erkennen, die durch die Analyse der spektralen Bestandteile beschreibbar und messbar sind. Die Spektralanalyse findet heute in nahezu allen Fachgebieten Anwendung. Es können damit Aussagen über die Beschaffenheit von Sternen, Gebäuden, Materialien, chemischen Stoffen, etc. getroffen werden, aber auch Prognosen über zukünftige Entwicklungen, wie z.B. über das Wetter oder ökonomische Trends, erstellt werden.

Auch aus der Audiosignalverarbeitung ist die Spektralanalyse nicht mehr wegzudenken. Nicht nur die naheliegende Ermittlung der spektralen Bestandteile ist ein wichtiges Einsatzgebiet, auch Spracherkennung, Rauschunterdrückung, Quellkodierung, Tonhöhenerkennung und -korrektur und Synthese wären ohne Spektralanalyse nicht oder nur eingeschränkt möglich.

2.2 Fourier-Reihe, -Analyse und -Transformation

Bereits im 18. Jahrhundert erkannte der französischen Mathematiker und Physiker Jean Baptiste Joseph Fourier (1768 - 1830), dass ein periodisches Signal in eine Summe von harmonischen Schwingungen und einem Offset zerlegt werden kann. [Sch05]

2.2.1 Fourier-Reihe

Stellt man den Satz von Fourier in mathematischer Form dar, so spricht man von der *Fourier-Reihe*, die wie folgt definiert ist [Sch05]:

Definition 2.1 (Fourier-Reihe)

$$f(t) = \frac{a_0}{2} + \sum_{n=1}^{\infty} a_n \cdot cos(n \cdot \omega_0 \cdot t) + \sum_{n=1}^{\infty} b_n \cdot sin(n \cdot \omega_0 \cdot t) \tag{2.1a}$$

$$\omega_0 = 2 \cdot \pi \cdot f_0 \,, \quad T_0 = \frac{1}{f_0} \tag{2.1b}$$

Die Zusammenhänge der verwendeten Symbole sind in Gleichung 2.1b angegeben. ω_0 stellt die Kreisfrequenz in $[\frac{rad}{s}]$ der Grundschwingung, f_0 die damit zusammenhängende Frequenz in $[Hz]$ und T_0 die Periodendauer in $[s]$ dar. Die Koeffizienten a_n, b_n werden im Zuge der *Fourier-Analyse* ermittelt, die im kommenden Abschnitt 2.2.2 beschrieben wird. Neben der hier dargestellten trigonometrischen Form, kann die Fourier-Reihe auch in komplexer Form mit dem Kern $e^{j \cdot n \cdot \omega_0 \cdot t}$ [Pap62] angeschrieben werden. Diese Form wird vorerst nicht verwendet, um die Funktionsweise einfacher darstellen zu können.

In Abbildung 2-1 wird anhand eines Rechtecksignals beispielhaft gezeigt, wie dieses mithilfe einer Summe endlich vieler Kosinusfunktionen angenähert werden kann. Je größer die Anzahl der Annäherungsschritte N, desto genauer wird das Ausgangssignal nachgebildet.

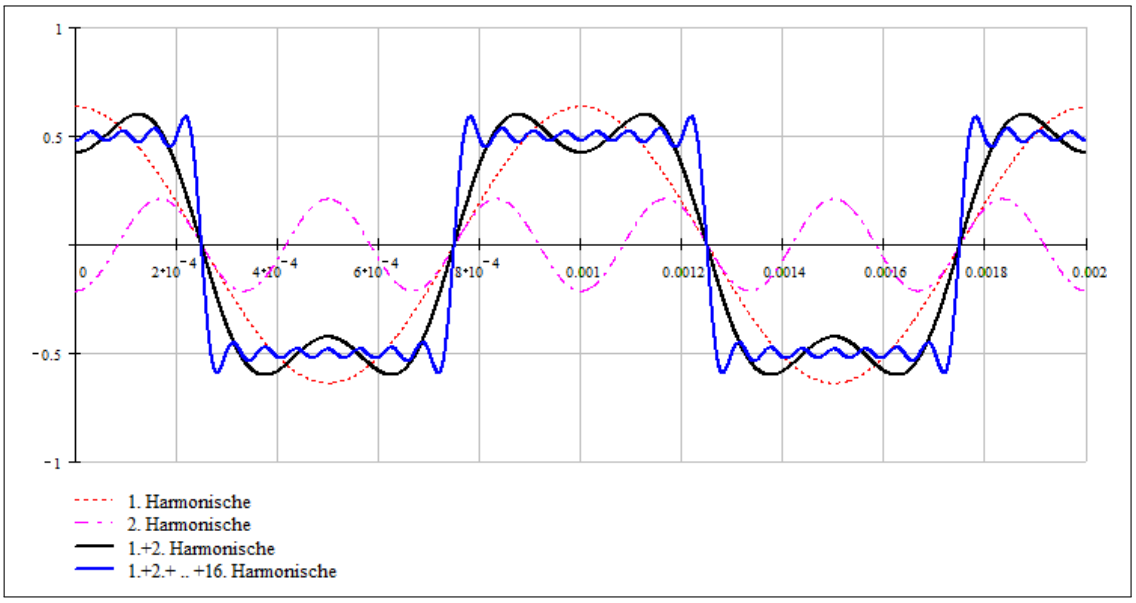

Abbildung 2-1: Fourier-Reihe eines Rechtecksignals (Mathcad™)

2.2.2 Fourier-Analyse

Das in Abbildung 2-1 gezeigte Beispiel setzt voraus, dass die Koeffizienten bereits bekannt sind. Sie geben darüber Auskunft, welche Parameter für die sinusförmigen Funktionen zu wählen sind, damit deren Fourier-Reihe das Ausgangssignal nachbildet. In Definition 2.2 wird die Berechnung der Koeffizienten a_n und b_n gezeigt. [Sch05]

Definition 2.2 (Fourier-Koeffizienten)

$$a_n = \frac{2}{T_0} \cdot \int_0^{T_0} f(t) \cdot cos(n \cdot \omega_0 \cdot t) dt \ , \quad n = 0, 1..\infty \tag{2.2a}$$

$$b_n = \frac{2}{T_0} \cdot \int_0^{T_0} f(t) \cdot sin(n \cdot \omega_0 \cdot t) dt \ , \quad n = 1, 2..\infty \tag{2.2b}$$

Zu den Parametern gehören Amplitude, Frequenz und Phase. Für die Frequenz kommen bei der Fourier-Reihe ausschließlich ganzzahlige Vielfache der Grundfrequenz $n \cdot \omega_0$ zum Einsatz. Die Phase wird durch das Vorzeichen der Amplitude (0°, 180°) und durch die Art des Koeffizienten bestimmt, wobei a_n für die Kosinus- (90°) und b_n für die Sinusfunktionsanteile (0°) steht. Letztere stehen im direkten Zusammenhang mit der Symmetrie der zu entwickelnden Funktion. Ist diese *gerade*, also zur Y-Achse spiegelsymmetrisch, dann gilt: $b_n = 0 \, \forall n$ Ist die zu entwickelnden Funktion dagegen *ungerade*, also gespiegelt um den Ursprung, dann gilt: $a_n = 0 \, \forall n$.
a_0 stellt als Anteil bei $\omega = 0$ eine Sonderform dar, die als *Gleichanteil* oder auch *Offset* bezeichnet wird und die vertikale Verschiebung der Funktion $f(t)$ widerspiegelt. [Sch05]

2.2.3 Integraltransformationen

Nach welchem Prinzip funktioniert die Ermittlung der Koeffizienten? Die Gleichungen 2.2a und 2.2b basieren auf der mathematischen Grundform der Integraltransformation, die wie folgt definiert ist [PM08]:

Definition 2.3 (Integraltransformation)

$$\tilde{f}(\lambda) = \int_a^b \phi(x, \lambda) \cdot f(x) \, dx \tag{2.3}$$

Bei $\tilde{f}(\lambda)$ handelt es sich um die Transformierte der Funktion $f(t)$. $\phi(x, \lambda)$ stellt den sog. *kernel* - zu deutsch *Kern* - der Transformation dar [PM08]. Diese elementare Grundform spielt auch später bei der Untersuchung der Wavelet-Transformation eine wichtige Rolle.

In Abbildung 2-2 wird anhand der Berechnung der Fourier-Koeffizienten gezeigt, nach welchem Prinzip Integraltransformationen arbeiten. Nach Gleichung 2.3 wird das zu analysierende Rechtecksignal ① mit der als Transformationskern dienenden Kosinusfunktion ② multipliziert, und anschließend die Fläche unter diesem Produkt ③ für die Periodendauer ermittelt. Als Parameter dienen hier die Frequenz, die einem ganzzahli-

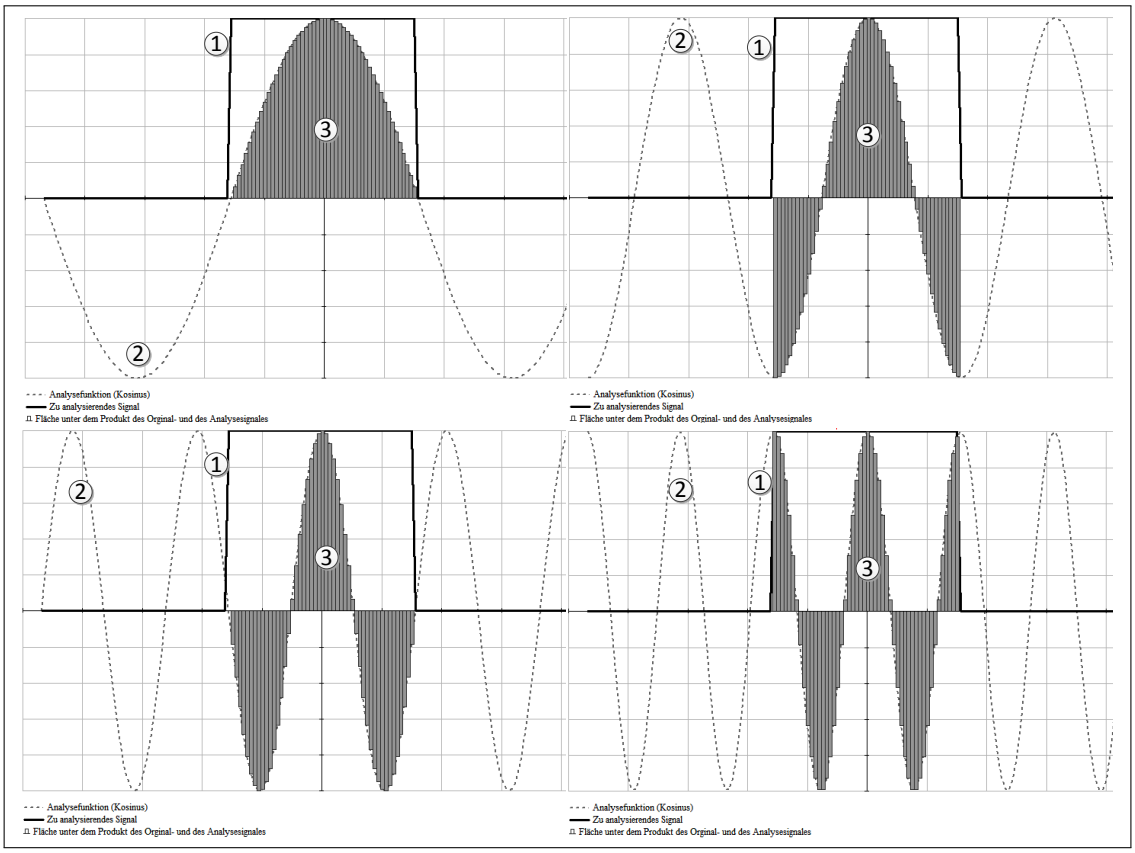

Abbildung 2-2: Fourieranalyse eines Rechtecksignals (Mathcad)

gen Vielfachen der Grundfrequenz entspricht, und die Phase, die durch das Vorzeichen bzw. durch die getrennten Koeffizienten a_n und b_n repräsentiert wird. Das Rechtecksignal wurde in diesem Beispiel zur Vereinfachung so gewählt, dass das Produkt der beiden Funktionen immer gleich der Kosinusfunktion im Intervall $t \in [\frac{-T_0}{2}, \frac{+T_0}{2}]$ ist:

$$f(t) = f_{Rechteck}(t) = \begin{cases} 1 & \forall\, t \in [\frac{-T_0}{4}, \frac{+T_0}{4}] \\ 0 & otherwise \end{cases}$$

Folgerung 2.4 (Interpretation des Analyseverfahrens von Integraltransformationen)
Um das Analyseverfahren der Integraltransformationen informell zu umschreiben, könnte man sagen, man probiere für jede Analysefunktion aus, wie gut sie in das zu analysierende Signal hineinpasst.

2.2.4 Fourier-Spektrum

Betrachtet man die ermittelten Koeffizienten gesondert, so repräsentieren sie das Spektrum des Signals. Da aufgrund der Analyse periodischer Signale nur ganzzahlige Vielfache der Grundfrequenz enthalten sind, wird das Spektrum als *diskretes Spektrum* oder auch *Linienspektrum* bezeichnet [Sch05]. Die folgende Abbildung 2-3 zeigt die Koeffizi-

enten eines Rechtecksignals dargestellt als Linienspektrum.

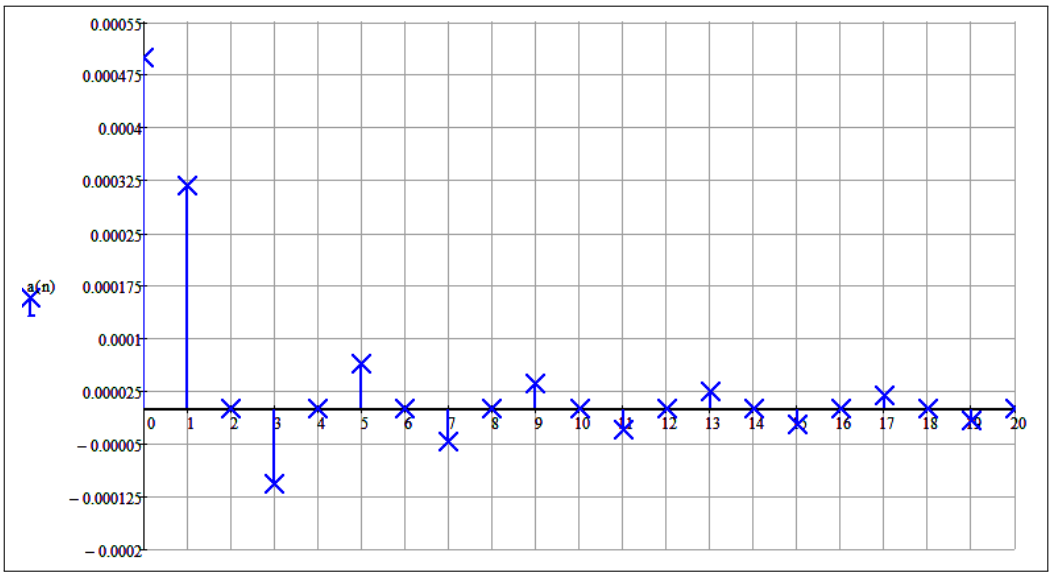

Abbildung 2-3: Fourier-Koeffizienten eines Rechtecksignals (Mathcad)

Oft wird auch nur der Betrag des Spektrums dargestellt Dabei werden die negativen Spektrallinien in Abbildung 2-3 auf die positive Y-Achse gespiegelt. Der Informationsverlust bezüglich der Vorzeichen müsste dann aber, für eine vollständig korrekte Darstellung, durch ein separates Phasendiagramm ausgeglichen werden. Darauf wird aber in der Regel verzichtet, da die Kerninformation für die meisten praktischen, technischen Anwendungen bereits im Betragsspektrum enthalten ist.

2.2.5 Orthogonalität

Bisher wurde geklärt, wie sich ein periodisches Signal in seine spektralen Bestandteile zerlegen und anschließend wieder zusammensetzen lässt. Aber warum funktioniert das? Weshalb gibt es nur eine Lösung für das Spektrum? Könnte man nicht mit einer anderen spektralen Zusammensetzung zum gleichen Ziel kommen, indem man z.B. nach ein paar ungenau bestimmten ersten Harmonischen den Fehler durch die folgenden wieder korrigiert?

Die vorweggenommene Antwort auf diese Fragen liefert eine spezielle Eigenschaft, welche jene Funktionen zueinander aufweisen, die als Transformationskern eingesetzt werden. Diese Eigenschaft wird als *Orthogonalität* bezeichnet. Betrachten wir zunächst die Definition 2.5 aus der Vektorgeometrie:

Definition 2.5 (Orthogonalität von Vektoren)
In der Geometrie werden zwei Vektoren mit einer Länge ungleich Null als *orthogonal* oder *rechtwinklig* zueinander bezeichnet, wenn deren Skalarprodukt gleich Null ist.

Gleichung 2.4 zeigt verschiedene Schreibweisen und Berechnungsvorschriften des Skalarprodukts, Gleichung 2.5 die Orthogonalitätsbedingung für Vektoren. Man kann aus ihnen erkennen, dass das Skalarprodukt immer dann Null wird bzw. *verschwindet*, wenn $cos(\varphi) = 0$ wird, was wiederum bei $\varphi = 90°, 270°, \ldots$ eintritt. Damit erklärt sich, warum man die Vektoren als *rechtwinklig* zueinander ansieht. [Pap09]

$$< \vec{a}, \vec{b} > := \vec{a} \cdot \vec{b} := |\vec{a}| \cdot |\vec{b}| \cdot cos(\varphi) = a_x \cdot b_x + a_y \cdot b_y \tag{2.4}$$

$$< \vec{a}, \vec{b} > \Big|_{|\vec{a}| \neq 0,\ |\vec{b}| \neq 0} = 0 \tag{2.5}$$

Diese Eigenschaft von Vektoren lässt sich auch auf Funktionen übertragen. Dabei geht man von der Betrachtung des *Vektorraums* $\mathbb{R}^2$ in die Betrachtung eines *Funktionenraums* über. Während der *Vektorraum* $\mathbb{R}^2$ eine Menge reellwertiger Paare (x und y bzw. Betrag und Phase) umfasst, beinhaltet der *Funktionenraum* eine Menge von Funktionen. Ein spezieller Raum, für den ebenso wie für Vektoren ein Skalarprodukt definiert ist, ist der *Hilbert-Raum* L^2. Definition 2.6 zeigt das Skalarprodukt für den Hilbert-Funktionenraum [Bän05].

Definition 2.6 (Skalarprodukt im Hilbert-Funktionenraum)

$$< f, g > \Big|_{f,\ g\ \in\ L^2(\mathbb{R})} := \int \overline{f(t)} \cdot g(t)\, dt \tag{2.6}$$

Für periodische Funktionen kann das unbestimmte Integral durch ein bestimmtes über die Periode T ersetzt werden. Die komplex Konjugierte $\overline{f(t)}$ bezweckt, dass auch bei komplexwertigen Funktionen das Skalarprodukt $< f, f >$ eine positive reelle Zahl ergibt. [Bän05]

Um einen Funktionenraum als Hilbert-Raum L^2 einordnen zu können, müssen dessen Funktionen gewisse Bedingungen erfüllen. Zu ihnen zählt auch die *Quadratintegrierbarkeit*, die in Definition 2.7 zu sehen ist. In praktischen Anwendungsfällen ist diese Bedingung immer erfüllt. Man spricht dort auch von Signalen endlicher Energie. [Bän05]

Definition 2.7 (Quadratintegrierbarkeit)

$$\int |f(t)|^2\, dt < \infty \tag{2.7}$$

Auf weitere Ausführungen zu Räumen und anderen mathematischen Hintergründen

wird an dieser Stelle verzichtet. Für einen tieferen Einblick in diese Materie sei hier auf [PM08] verwiesen.

Mit Hilfe des in Definition 2.6 gezeigten Skalarprodukts kann die Orthogonalität zweier Funktionen geprüft werden. Angewandt auf den Transformationskern kann somit nachgewiesen werden, wie es um die Eindeutigkeit der Transformationskoeffizienten steht. Im Beispiel 2.8 wird gezeigt, dass die Ermittlung der Koeffizienten im Zuge der Fourieranalyse auf einem orthogonalen Transformationskern basiert. [Sch05]

Beispiel 2.8 (Orthogonalität des Fourieranalyse-Transformationskerns)

$$\int_0^{T_0} sin(n\,\omega_0\,t)\cdot sin(m\,\omega_0\,t)\;dt = \begin{cases} 0 & \text{für } n \neq m \\ \frac{T_0}{2} & \text{für } n = m \end{cases} \tag{2.8a}$$

$$\int_0^{T_0} cos(n\,\omega_0\,t)\cdot cos(m\,\omega_0\,t)\;dt = \begin{cases} 0 & \text{für } n \neq m \\ \frac{T_0}{2} & \text{für } n = m \end{cases} \tag{2.8b}$$

$$\int_0^{T_0} sin(n\,\omega_0\,t)\cdot cos(m\,\omega_0\,t)\;dt = 0 \tag{2.8c}$$

Zusammenfassend könnte man die eingangs gestellten Fragen mit Hilfe des Beispiels 2.8 folgendermaßen zusammenfassen:

Folgerung 2.9 (Orthogonale Basis)
Wenn man die Funktionen des Kerns mit der Transformation analysiert, zu der sie gehören, ist das Ergebnis immer Null, es sei denn, man analysiert sie mit sich selbst. Ist diese Aussage ausnahmslos für alle Bestandteile des Transformationskerns zutreffend, so spricht man von einer ***orthogonalen Basis****. Diese bewirkt, dass die Transformation eindeutige Ergebnisse liefert.*

2.2.6 Fourier-Transformation

Bisher haben wir uns auf die Analyse periodischer Signale beschränkt. In der Praxis wird man aber öfter mit nicht-periodischen Signalen konfrontiert. Die menschliche Stimme besteht z.B. aus komplexen, sich naturbedingt niemals wiederholenden Teilen und ist nicht, wie ein sinusförmiges Signal, vorhersehbar. Wie können ihre spektralen Bestandteile analysiert werden?

Für die Lösung des Problems behilft man sich eines Tricks:
Die Periode T_0 des Signals wird so lange vergrößert, bis sie gegen ∞ geht. Angewandt auf die Fourieranalyse geht damit das Integral über die Periode T_0 in ein unbestimmtes

Integral von $-\infty$ bis ∞ über. Die diskret indizierten Koeffizienten a_n, b_n mit $n \in \mathbb{N}$ gehen über in eine kontinuierliche Funktion $\mathbb{R} \mapsto \mathbb{C}$, da $\omega_0 = \frac{2\pi}{T_0}$ gegen Null geht. Schlussendlich wird aus der Summe der Fourier-Reihe ein Integral. Die Fourier-Transformation und die inverse Fourier-Transformation werden damit folgendermaßen definiert [Pap62]:

Definition 2.10 (Fourier-Transformation 2.9 und inverse Fourier-Transformation 2.10)

$$F(\omega) = \int_{-\infty}^{\infty} f(t) \cdot e^{-j\,\omega t}\, dt \tag{2.9}$$

$$f(t) = \frac{1}{2\pi} \cdot \int_{-\infty}^{\infty} F(\omega) \cdot e^{j\,\omega t}\, d\omega \tag{2.10}$$

Neben Definition 2.10, die als *asymmetrische Form der Fourier-Transformation* bezeichnet wird und im Ingenieurwesen üblich ist, findet man in der mathematischen Literatur (siehe unter anderem [PM08]), auch häufig den Faktor $\frac{1}{\sqrt{2\pi}}$ in beiden Integralen vor, welches der *symmetrischen Form der Fourier-Transformation* entspricht.

Das kontinuierliche Spektrum, das hier entsteht, wird auch als *Amplitudendichtespektrum* bezeichnet, da dessen Werte die Einheit $[\frac{Amplitude}{Hz}]$ aufweisen. Bei elektrischen Spannungen wäre das z.B. $[\frac{V}{Hz}]$. In Abbildung 2-4 ist das Amplitudendichtespektrum eines Rechtecksignals, das in diesem Fall als aperiodisch angenommen wird, dargestellt. [Mey08]

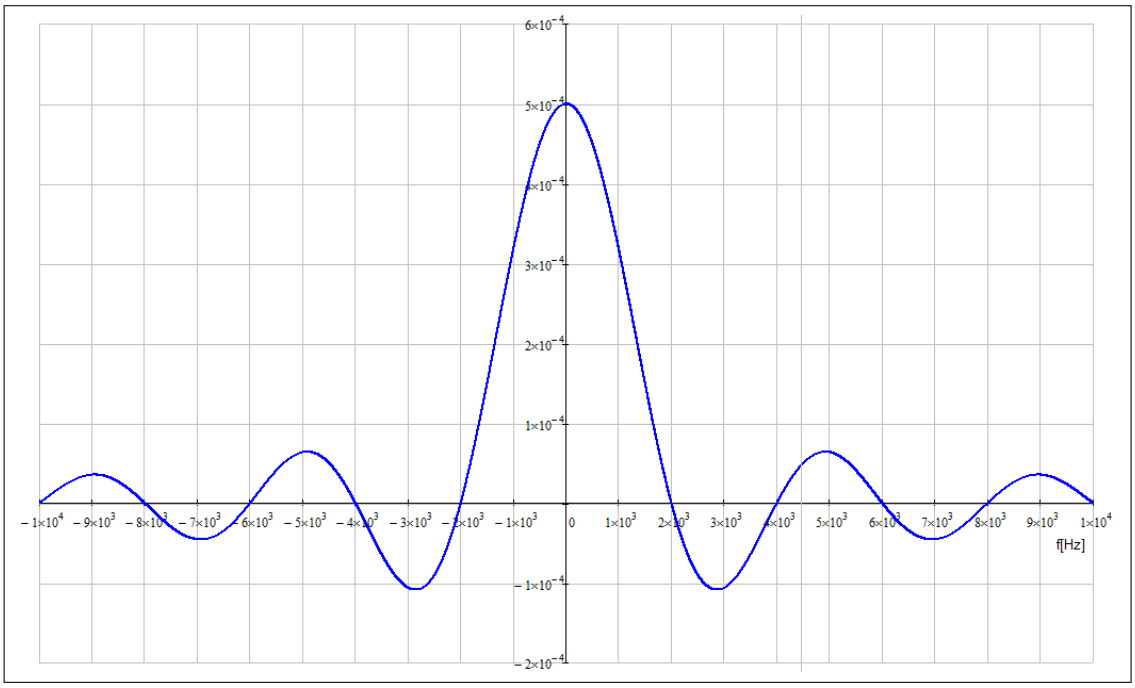

Abbildung 2-4: Fourier-Transformierte eines Rechtecksignals (Mathcad)

Die entstandene Spektralfunktion, deren Hauptbestandteil $\frac{sin(x)}{x}$ als *Spaltfunktion*, *si-* oder auch *sinc-Funktion* bezeichnet wird, lässt sich wie folgt anschreiben:

$$F(\omega) = A \cdot \frac{T_0}{2} \cdot \frac{sin\left(\omega \frac{T_0}{4}\right)}{\omega \frac{T_0}{4}} \tag{2.11}$$

Auf den ersten Blick erscheint es ungewöhnlich, dass das Spektrum auch negative Frequenzen beinhaltet. Die Ursache dafür ist, dass es sich hierbei um eine Beschreibung in $\mathbb{C}$ handelt, bei der sich die komplexen Schwingungen $e^{j\,\omega t}$ sowie $e^{-j\,\omega t}$ zu einer reellen sinusförmigen Funktion überlagern. [Sch05]

Wenn man die Abbildung 2-4 mit der Abbildung 2-3 der Fourier-Reihenkoeffizienten aus dem Abschnitt 2.2.2 vergleicht, kann man auch grafisch die Analogien zwischen den Spektren der Fourier-Reihe für periodische Signale und der Fourier-Transformation für aperiodische Signale erkennen: Die Fourier-Koeffizienten bestehen nur bei ganzzahligen Vielfachen der Grundschwingung, folgen aber dem Verlauf des Fourier-Transformationsspektrums. Auch die Nulldurchgänge sind in beiden Diagrammen identisch. Die Vergrößerung der Periode auf ∞ hat die Dichte der Linien so weit erhöht, dass eine kontinuierliche Funktion entstanden ist, bei der die Grundschwingung und der Gleichanteil bei $f \cong 0$ zu verschmelzen scheinen.

2.2.7 Diskrete Fourier-Transformation (DFT)

In den vorangegangenen Abschnitten wurde gezeigt, dass die Fourier-Analyse und deren Abwandlungen vielseitig und anpassungsfähig sind. Bisher wurden allerdings nur kontinuierliche, sog. *analoge* Signale analysiert. In der digitalen Signalverarbeitung stehen die Signale aber nicht in dieser, sondern in Form einer Folge von diskreten Abtastwerten zur Verfügung.
Kann das bisher Gewonnene auch auf abgetastete Signale übertragen werden?
Betrachten wir dazu zunächst das Abtasttheorem aus Definition 2.11 [Ant05], das die elementare Bedingung für die Darstellung kontinuierlicher Signale als Abtastwerte festlegt. T_s steht dabei für die Abtastperiode, die wiederum dem Reziprok der Abtastfrequenz $f_s = \frac{1}{T_s}$ entspricht.

Definition 2.11 (Abtasttheorem)
Ein bandbegrenztes Signal $f(t)$, für das

$$F(j\,\omega) = 0 \quad bei \quad |\omega| \geq \frac{\omega_s}{2} \quad und \quad \omega_s = \frac{2\pi}{T_s} \tag{2.12}$$

gilt, kann durch die Abtastwerte $f(n \cdot T_s)$ eindeutig bestimmt werden. [Ant05]

Folgerung 2.12 (Interpretation des Abtasttheorems)
Wenn das Abtasttheorem eingehalten wird, die Abtastung also mit mehr als dem doppelten der höchsten im Signal $f(t)$ enthaltenen Frequenz erfolgt, dann kann das ursprüngliche Signal wieder eindeutig aus den Abtastwerten rekonstruiert werden. Somit ist sichergestellt, dass die Abtastwerte den notwendigen Informationsgehalt haben, um das kontinuierliche Signal im diskreten Bereich „vertreten" zu können.

Ist damit auch eine Fourier-Transformation im diskreten Wertebereich möglich?
Ja, wobei es für diese noch einiger Anpassungen bedarf: Die kontinuierliche Zeitvariable t wird durch die diskreten Zeitschritte $n \cdot T_s$ ersetzt, das Differential dt durch das Abtastintervall T_s und das unbestimmte Integral durch eine Summe von $n = -\infty$ bis ∞. Das Abtastintervall T_s stellt einen konstanten Faktor vor der Summe dar, der je nach Form auch weggelassen wird, sofern dass bei der inversen Transformation dementsprechend berücksichtigt wird.

Da eine unendliche Abtastfolge nicht praktikabel ist, wird sie auf eine endliche Anzahl N begrenzt, was auch als *Windowing* - zu deutsch Fensterung - bezeichnet wird. Diese Einschränkung hat einen Einfluss auf das Ergebnis der Spektralanalyse und wird in Abschnitt 2.3 genauer erläutert. Die bisher geschilderten Anpassungen zusammengefasst haben folgende, in Gleichung 2.13 angeschriebene Zwischenform der DFT zur Folge [Grü08]:

$$X(f) = \sum_{n=0}^{N-1} x(nT_s) \cdot e^{-j2\pi f n T_s} \tag{2.13}$$

Diese f_s-periodische Funktion hat, ohne näher auf den Beweis einzugehen, nur an N Frequenzstellen Funktionswerte, d.h. sie liefert für N Abtastwerte ein frequenzdiskretes Linienspektrum mit N Spektralwerten. Berücksichtigt man diese Eigenschaft, kann sie auch als Folge von $k = 0, 1, 2, \ldots, N-1$ Werten angeschrieben werden. In diese Form gebracht entspricht die Gleichung schlussendlich der Definition der DFT. Nach dem gleichen Herleitungsprinzip wird auch die inverse DFT, kurz IDFT, definiert. Zusammen bilden sie das sog. *diskrete Fourier-Transformationspaar*, wie es in Definition 2.13 angeführt ist. [Grü08]

Definition 2.13 (DFT (2.14) und IDFT (2.15))

$$X_k = \sum_{n=0}^{N-1} x_n \cdot e^{-jkn\frac{2\pi}{N}} \tag{2.14}$$

$$x_n = \frac{1}{N} \cdot \sum_{k=0}^{N-1} X_k \cdot e^{jkn\frac{2\pi}{N}} \tag{2.15}$$

Zur Veranschaulichung ist in Abbildung 2-5 das Ergebnis einer DFT mit $N = 32$ eines

abgetasteten Rechtecksignals dargestellt. Deutlich zu erkennen ist die Symmetrie der DFT. Die Werte sind um $\frac{N}{2}$ gespiegelt. Diese Spiegelachse entspricht dabei gleichzeitig der halben Abtastfrequenz $\frac{f_s}{2}$ und damit der höchst möglichen Frequenz im analysierten Signal. Die Redundanz kommt aus dem gleichen Grund zustande, aus dem auch bei der Fourier-Transformation positive und negative Frequenzen im Ergebnis enthalten sind. In der Praxis bedeutet das, dass nur die Daten bis $\frac{N}{2}$ verwendet werden. [Grü08]

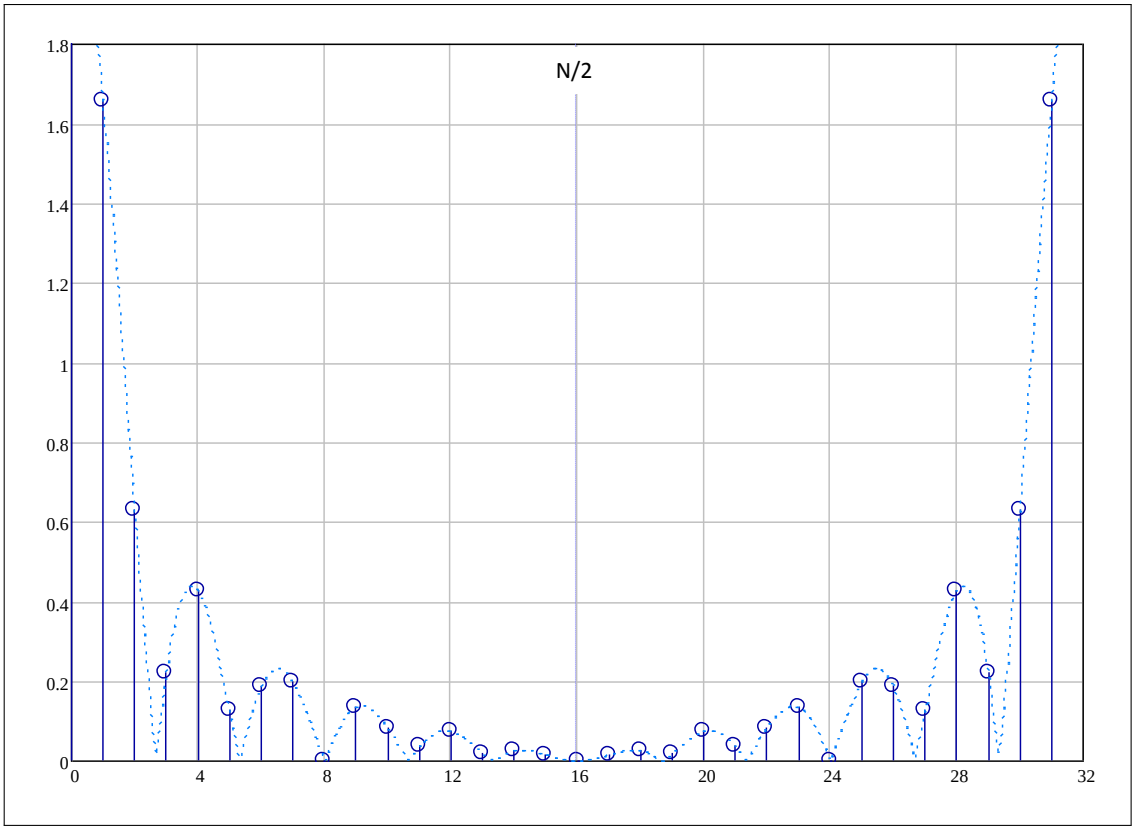

Abbildung 2-5: DFT eines abgetasteten Rechtecksignals (Mathcad)

2.2.8 Fast Fourier Transformation (FFT)

Mit der DFT ist es, wie im vorangegangenen Abschnitt 2.2.7 erläutert, möglich, eine Fourier-Transformation mittels digitaler Signalverarbeitung umzusetzen. An den Rechner werden dabei aber sehr hohe Ansprüche gestellt: N diskrete Werte, üblicherweise im Fließkommaformat, müssen über N komplexe Multiplikationen und $(N-1)$ Additionen verarbeitet werden. Für hohe N bedeutet das einen sehr hohen Aufwand, da die Anzahl der Fließkommaoperationen quadratisch zunimmt. 1965 haben *Cooley/Tukey* festgestellt, dass die DFT einen hohen Anteil an Redundanz aufweist und entwickelten einen Algorithmus zur schnellen Berechnung der DFT, der als *Fast Fourier Transformation* - kurz FFT - bekannt ist. In anderen Anwendungsbereichen spricht man auch vom *Cooley-Tukey-Verfahren*. [Ant05]

Das Grundprinzip dabei ist, dass sich die Berechnung der Summe von N Summanden in die Berechnung zweier getrennter Summen mit jeweils nur halb so vielen Summanden aufteilen lässt. Dabei wird jeder zweite Abtastwert weggelassen, wobei für die erste Summe alle geradzahligen und für die zweite Summe alle ungeradzahligen n eingesetzt werden. Gleichung 2.16 zeigt das Prinzip [Ant05]:

$$X_k = \sum_{n=0}^{\frac{N}{2}-1} x_{(2n)} \cdot e^{-jk(2n)\frac{2\pi}{N}} + \sum_{n=0}^{\frac{N}{2}-1} x_{(2n+1)} \cdot e^{-jk(2n+1)\frac{2\pi}{N}} \tag{2.16}$$

Bis auf die veränderten Indizes entspricht jeder der beiden neu entstandenen Terme wieder der DFT, jetzt allerdings nur mehr mit $\frac{N}{2}$ Summanden. Es lässt sich erkennen, dass die Aufteilung rekursiv auf die neu entstandenen Terme angewandt werden kann. Auf Gleichung 2.16 bezogen würden dann also im nächsten Schritt 4 Terme mit jeweils einer Summe über $\frac{N}{4}$ Summanden entstehen usw. Nach einer Anzahl von $\frac{N}{2}$ solcher Rekursionen, wenn also die Summe nur mehr aus einem Summanden besteht und damit wegfällt, ist die unterste Rekursionsebene erreicht. Abbildung 2-6 zeigt das Prinzip der FFT grafisch anhand eines Beispiels mit $N = 8$. [Ant05]

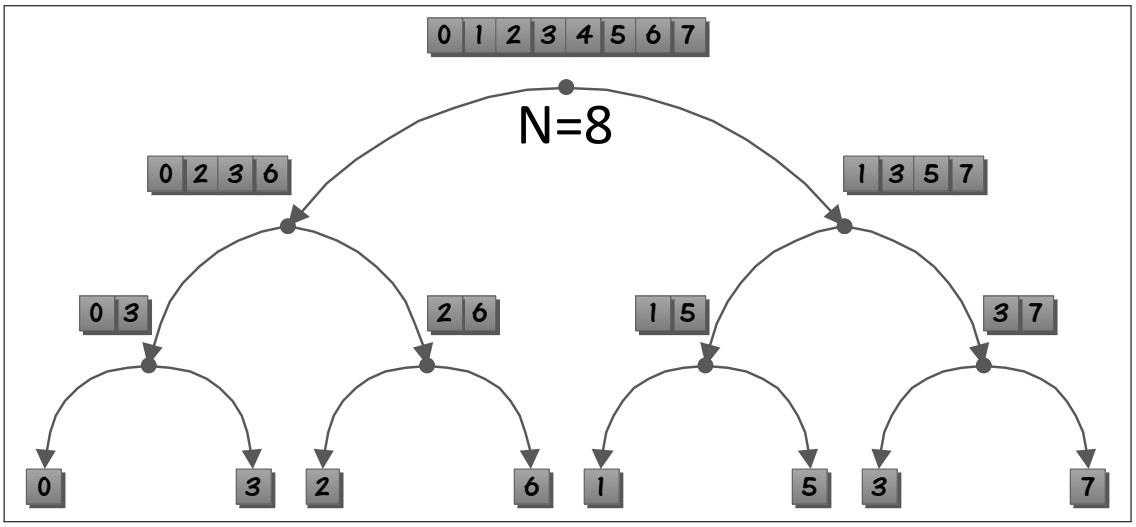

Abbildung 2-6: Prinzip der FFT

Die FFT hat zur Folge, dass die Anzahl der Multiplikationen von N^2 auf $\frac{N}{2} \cdot log_2(N)$ reduziert wird. Auf das Beispiel mit $N = 8$ bezogen bedeutet das, dass nur mehr 12 statt 64 Multiplikationen notwendig sind. Voraussetzung dafür ist allerdings, dass N einer Potenz von 2 entspricht. [Ant05]

Unter dem mittlerweile etwas weitläufiger verwendeten Begriff FFT werden in der Praxis sämtliche leistungssteigernde Maßnahmen zusammengefasst. Dabei werden neben anderen algorithmischen Maßnahmen, auf die hier nicht näher eingegangen wird, auch technische Optimierungen eingesetzt. Zu diesen zählen u.a. die Verwendung spezieller Prozessorbefehlssätze wie z.B. SSE, Parallelverarbeitung, etc.

2.2.9 Zusammenfassung der Eigenschaften

Tabelle 2-1 zeigt zusammenfassend die wichtigsten Eigenschaften der bisher behandelten Transformationen:

Transformation	Zeitfunktion	Spektrum
Fourier-Reihe	kontinuierlich, periodisch	frequenzdiskret
Fourier-Transformation	kontinuierlich, aperiodisch	kontinuierliche Amplitudendichte
DFT, FFT	diskret, gefenstert	diskret, f_s-periodisch

Tabelle 2-1: Fourier-Transformationen zusammengefasst

2.3 Fensterung

Wie in Abschnitt 2.2.8 bereits erwähnt, ist es in der praktischen, digitalen Signalverarbeitung nicht möglich, eine Transformation, wie z.B. die DFT, über unendlich viele Abtastwerte durchzuführen. Auch dann, wenn das zu analysierende Ausgangsmaterial in endlicher Länge vorliegt, wie z.B. bei einem Musikstück, ist das daraus resultierende Gesamtspektrum wenig informativ, da es keinerlei Aussage über den zeitlichen Verlauf zulässt. Bei periodischen Signalen wäre die Ermittlung eines Spektrums mit einer endlichen Anzahl $N = k \cdot \frac{T_0}{T_s}$, $k \in \mathbb{N}^+$ möglich, allerdings sind Signale dieser Art selten. [Grü08]

Dem Problem kann nur mit einem Kompromiss begegnet werden:
Es wird ein festgelegter, endlicher Abschnitt von Abtastwerten, wie durch ein Fenster betrachtet, analysiert. Dieser Vorgang, der unter der Bezeichnung *Fensterung* oder auch *windowing* bekannt ist, wird anschließend für die verbleibenden Abschnitte wiederholt. Die gefensterte DFT wird dabei auch als *Short Time Discrete Fourier Transformation*, kurz STDFT, bezeichnet. [Ant05]

Das Abschneiden von Abtastwerten bleibt aber nicht ohne Folgen für das daraus resultierende Spektrum. Aus Sicht der Transformation wird eine endliche Anzahl N als periodisches Zeitsignal angesehen und somit angenommen, es würde sich in genau dieser Weise periodisch fortsetzen. Um sich das besser vorstellen zu können, kann man es auch mit den Auswirkungen auf den Klang vergleichen, die ein *hängender* CD-Player hat, der aufgrund eines Lesefehlers immer wieder den in den Speicher eingelesenen Teil wiederholt. Oft sind zu Beginn eines Wiederholvorgangs hochfrequente Spitzen zu hören, die durch das Abschneiden des Signals abseits von dessen Nulldurchgängen zustande kommen.

Auf das Spektrum zeigt die Fensterung nun folgende Auswirkungen [Grü08]:

- **Verschmierung**: Statt einer Spektrallinie entstehen mehrere benachbarte Linien, wenn im Signal Schwingungen enthalten sind, die nicht exakt in das *Frequenzraster* des Spektrums passen.
- **Aliasing**: Auch wenn die Abtastfrequenz f_s so gewählt wurde, dass das Abtasttheorem erfüllt ist, so muss das nicht zwangsläufig bedeuten, dass das auch für das gefensterte Signal gilt. [Hof97]
- **Leck-Effekt(Leakage)**: Außerhalb des Hauptlappens entstehen neue Spektrallinien, die hauptsächlich aufgrund des abrupten Übergangs zur periodisch fortgesetzten Abtastfolge entstehen. [KK06]

Die Fehler lassen sich durch eine höhere Auflösung N verringern. Der Leck- und der Aliasing-Effekt lassen sich zusätzlich durch die Wahl einer geeigneteren *Fensterfunktion* reduzieren.

Bisher wurde ausschließlich von einem sog. *Rechteckfenster* ausgegangen, bei dem der zu analysierende Ausschnitt dem Signal durch einen harten Schnitt unverändert entnommen wird. Wählt man einen sanfteren Übergang, wie er z.B. auch in der Audio- und Videoschnitttechnik durch sog. *Fade-Ins/Outs* zum Einsatz kommt, so wirkt sich das auch positiv auf das Spektrum aus, insbesondere hinsichtlich des Leck-Effekts. Allerdings müssen dazu die Abtastwerte an den Fenstergrenzen verändert werden, wodurch das Originalsignal und damit auch dessen Spektrum verändert wird. [Par10]

Folgerung 2.14 (Auswirkungen von Zeitfenstern auf das Frequenzspektrum)
Je „weicher“ die Fensterfunktion ist, desto geringer wird der Einfluss des Leck-Effekts. Damit setzt sich die spektrale Darstellung des analysierten Signals klarer von den Störanteilen durch die Begrenzung von N ab. Die Breite des Hauptlappens nimmt dabei aber zu, wodurch die enthaltenen Frequenzen weniger genau bestimmt werden können und somit die effektive spektrale Auflösung abnimmt.

Abbildung 2-7 zeigt die Auswirkungen verschiedener Fenster am Beispiel einer DFT eines mit $f_s = 200Hz$ abgetasteten Sinussignals $f_{sinus} = 5Hz$. Deutlich zu erkennen sind der sehr hohe Einfluss des Leck-Effektes bei einer Rechteckfensterung, sowie der breitere Hauptlappen bei *weichen* Fenstern, die zu einer wesentlich breiteren Darstellung des eigentlich als dünne Linie erwarteten Sinussignals führt.

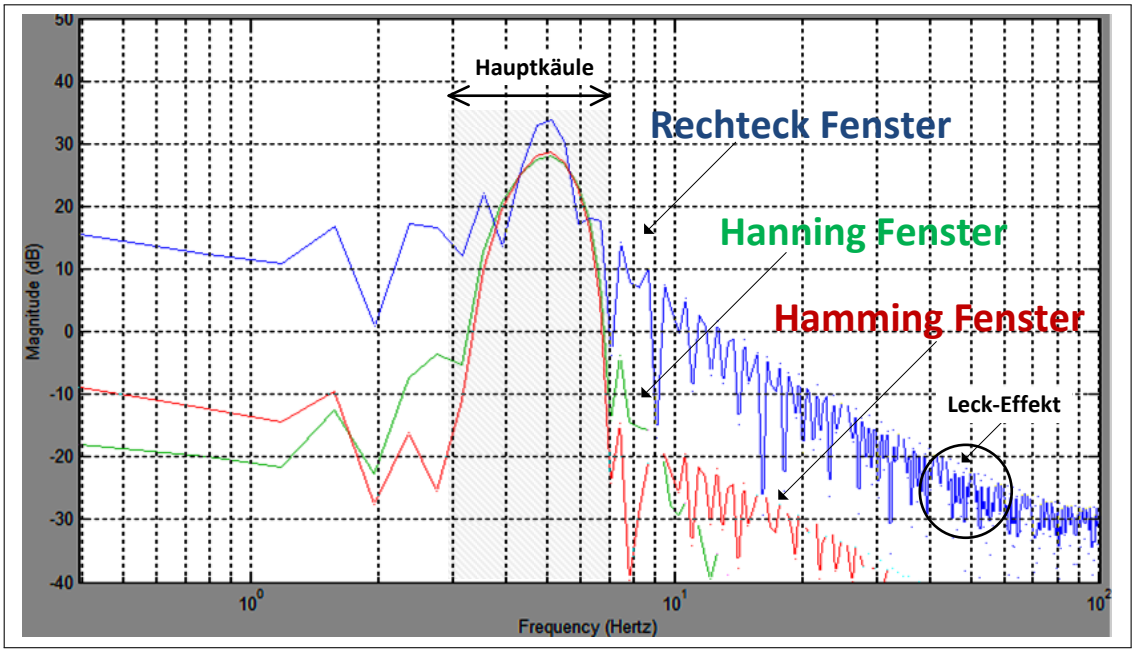

Abbildung 2-7: DFT eines Sinussignals unter Einsatz verschiedener Fenstern (Matlab™)

Vom mathematischen Blickwinkel aus betrachtet, wird die Abtastfolge mit der Fensterfunktionsfolge im Zeitbereich multipliziert. Das ist wiederum gleichbedeutend mit einer Faltung im Frequenzbereich [Hof97]. Auf diese Weise lassen sich die Auswirkungen verschiedener Fenster auf das Spektrum genau bestimmen und das für das jeweilige Einsatzgebiet am besten geeignete Fenster auswählen. Ohne im Detail auf die Berechnung einzugehen, zeigt Tabelle 2-2 die wichtigsten Spezifikationen einiger Fensterfunktionen im Vergleich. Auch hieraus ist ersichtlich, dass ein Kompromiss zwischen Hauptlappenbreite und Seitenlappenamplitude eingegangen werden muss.

Fenster	Breite des Hauptlappens	Amplitude des höchsten Seitenlappens [dB]	Minimale Sperrdämpfung [dB]
Rechteck	$4\pi/N$	-13	-21
Hanning	$8\pi/N$	-31	-44
Hamming	$8\pi/N$	-43	-53
Blackman	$12\pi/N$	-57	-74

Tabelle 2-2: Fensterfunktionen [MW99]

2.4 Wavelets

2.4.1 Einführung

Wie in den vorangegangenen Abschnitten beschrieben, muss bei der DFT *getrickst* werden, damit nicht-stationäre Signale überhaupt analysiert werden können und ein Zeitbezug hergestellt werden kann. Warum eignen sich die DFT, aber auch die anderen Mitglieder der Fourier-Transformationsfamilie, nicht für diese Art von Signalen?

Die Antwort darauf kann gefunden werden, wenn man den Kern dieser Transformationen genauer betrachtet: Es fällt auf, dass er ausschließlich sinusförmige Funktionen enthält. Deren Rolle könnte man als *Baukasten* zum Nachbau der zu analysierenden Funktion beschreiben. Der sinusförmige Kern mag in der komplexen Exponentialschreibweise nicht sofort ins Auge stechen, kann aber auch hier mit Hilfe der *Euler'schen Formel* [Ant05], die in Gleichung 2.17 dargestellt ist, nachgewiesen werden:

$$e^{j\varphi} = cos(\varphi) + j \cdot sin(\varphi) \tag{2.17}$$

Der sinusförmige Kern hat einen erheblichen Nachteil: Er erstreckt sich über den gesamten Zeitbereich und ist damit zeitlich nicht lokalisiert. Diese Eigenschaft wird der Transformation durch den Kern *aufgezwungen*, wodurch auch diese zeitlich nicht lokalisiert ist. Das hat z.B. auch zur Folge, dass sich die DFT nicht für Datenkompression eignet, da einerseits eine hohe Anzahl an Koeffizienten und somit Daten notwendig ist, um Signale mit lokalen Sprüngen nachzubilden. Andererseits ist der Fehler durch das Weglassen von Koeffizienten nicht lokal begrenzt, sondern erstreckt sich über den gesamten Zeitbereich und hätte somit einen wesentlichen Einfluss auf die Qualität des daraus wiedergewonnenen Signals zur Folge. [Bän05]

In Abbildung 2-8 wird am Beispiel eines mit 100 Fourier-Koeffizienten nachgebildeten, nicht-stationären Signals gezeigt, wie sich die Fehler auswirken. ① zeigt einen lokalen Sprung, der zur korrekten Nachbildung möglichst viele Koeffizienten fordert. ② zeigt die durchgehende, zeitlich nicht begrenzte *Welligkeit*, die durch die Begrenzung auf eine endliche Anzahl von Koeffizienten zustande kommt.
Was würde geschehen, wenn man den sinusförmigen Kern der Fourier-Transformation durch einen anderen, zeitlich begrenzten Kern ersetzen würde? Können dafür beliebige Funktionen verwendet werden? Welche Bedeutung haben die daraus resultierenden Koeffizienten?
Mit Fragen dieser Art setzt sich die *Wavelet*-Theorie auseinander. Das Wort *Wavelet* - zu deutsch *Wellchen* oder *kleine Welle* - ist aus dem französischen Wort „ondelette" entstanden, das teils sinngemäß, teils phonetisch in das Englische übersetzt wurde. [Bän05]

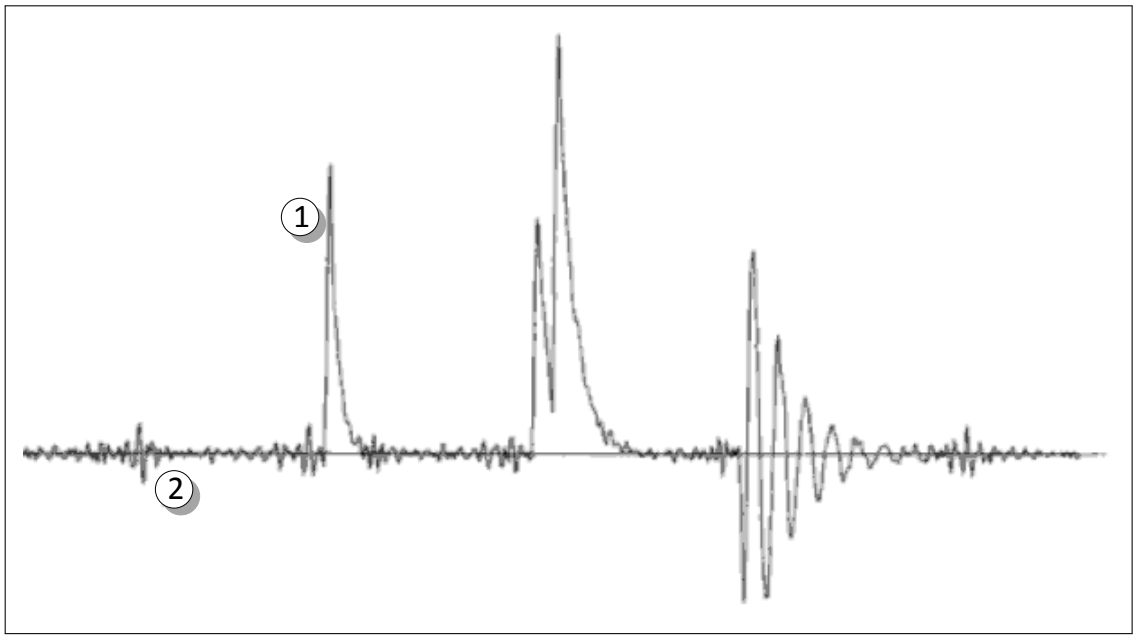

Abbildung 2-8: Rekonstruktion eines Signals mit 100 Fourier-Koeffizienten (Quelle: [Bän05])

2.4.2 Wavelet-Transformation

In Anlehnung an die Grundform der Integraltransformationen, wie sie in Definition 2.3 des Abschnitts 2.2.2 vorgestellt wurde, wird die kontinuierliche *Wavelet-Transformation* - kurz CWT - wie folgt definiert:

Definition 2.15 (Kontinuierliche Wavelet-Transformation (CWT) [1])

$$F^{\psi}(a,s) = |s|^{-1/2} \int_{-\infty}^{\infty} f(t) \cdot \psi\left(\frac{t-a}{s}\right) dt \tag{2.18}$$

Grundsätzlich ähnelt Definition 2.15 der Grundform der Integraltransformationen, da die tatsächliche Funktion des Kerns, repräsentiert durch den Platzhalter ψ, nach wie vor nicht festgelegt ist. Allerdings deutet das Argument $\frac{t-a}{s}$ darauf hin, dass gewisse Gesetzmäßigkeiten zwischen den Bestandteilen des Kerns gelten. Diese stellen das elementare Erkennungsmerkmal der Wavelet-Transformation dar: Ausgehend vom sog. *Mother-Wavelet*, das die Basisfunktion darstellt, werden alle weiteren Funktionen des Transformationskerns durch eine zeitliche Verschiebung um den Abstand $a \in \mathbb{R}$ und/oder einer Streckung mit dem Skalierungsfaktor $s \in \mathbb{R}^{\neq 0}$ aus dieser Basisfunktion gebildet [Dau92]. Den vorangestellten Faktor $|s|^{-1/2}$ könnte man auch als $\frac{1}{\sqrt{s}}$ anschreiben. Dieser ist bereits aus der Untersuchung der Symmetrie des Transformationspaares in

[1] Anmerkung: Die Definition wurde aus [Dau92] entnommen, jedoch unter der Verwendung der Variablenbezeichnungen von [Bän05], da s als Skalierungsfaktor naheliegender erschien als a. Des Weiteren wurde auf die Definition der inversen Form verzichtet, da sie hier für unsere Belange nicht von Bedeutung ist.

Abschnitt 2.2.6 bekannt und dient auch hier dem Energieerhalt, wodurch sichergestellt wird, dass bei aufeinanderfolgender Transformation und Rücktransformation die Amplitude des Signals unverändert bleibt. Der Vollständigkeit halber sei noch erwähnt, dass bei der Verwendung komplexwertiger Wavelets in der Definition 2.15 die komplex Konjugierte ψ^* verwendet werden müsste. [Mal99]

Abbildung 2-9 zeigt die grafische Darstellungsform der Koeffizienten und damit das Spektrum eines Beispielsignals, das im oberen Teil dargestellt ist:

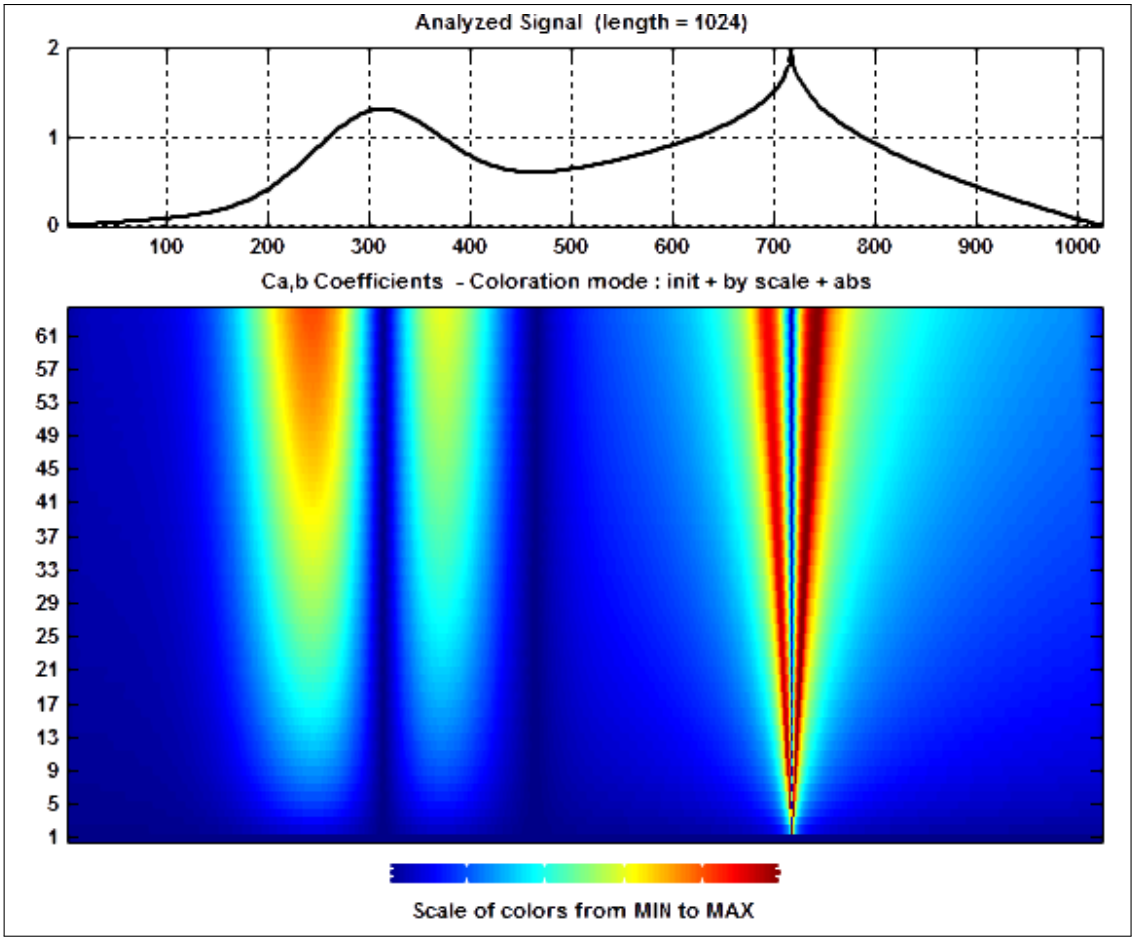

Abbildung 2-9: Koeffizienten der CWT (Matlab^{TM})

Die CWT eignet sich besonders gut, um lokale Merkmale zu erkennen. Sie wird daher u.a. auch in der Bildbearbeitung zur Kantenerkennung eingesetzt. Das lässt sich auch gut in Abbildung 2-9 erkennen. Im Vergleich zum Fourier-Spektrum ist eine deutlich bessere Zeitauflösung, speziell bei hohen Frequenzen, zu erkennen. Die spektrale Sicht auf das Signal stellt sich zudem als besonders gut lesbar heraus. [Bän05] Der Grund dafür ist allerdings, dass die Informationen hochgradig redundant vorliegen, worauf in Abschnitt 2.4.5 noch genauer eingegangen wird. Damit ist auch ein hoher Rechenaufwand verbunden, der insbesondere an Echtzeitanwendungen hohe Ansprüche stellen würde. [LMR98]

Ein zunächst etwas gewöhnungsbedürftiger Aspekt ist, dass die Frequenz nicht mehr direkt, sondern nur mehr indirekt durch den Skalierungsfaktor s repräsentiert wird. Das ist auch in Abbildung 2-9 zu sehen, bei der hohe Konzentrationen nahe der X-Achse immer an jenen Stellen zu beobachten sind, an denen im Zeitsignal hochfrequente Anteile aufgrund der Wellenform zu erahnen sind. Diese ist umgekehrt proportional zur Frequenz $s \sim \frac{1}{f}$. Das heißt, dass ein kleiner Skalierungsfaktor die Vorkommnisse hoher Frequenzanteile widerspiegelt und umgekehrt. [Add02]

2.4.3 Bedingungen an die Wavelet-Basisfunktionen

Bevor auf die Bildung einer diskreten Form und die Beseitigung der Redundanz der Wavelet-Transformation eingegangen werden kann, müssen zunächst die Wavelets genauer betrachtet werden. Wie im vorangegangenen Abschnitt 2.4.2 bemerkt, ist die Wavelet-Transformation nicht fest an eine bestimmte Basisfunktion gebunden, wie das z.B. bei der Fourier-Transformation der Fall ist. Vielmehr sind die Basisfunktionen, die einer skalierten und verschobenen Version des als Vorlage dienenden Mother-Wavelets entsprechen, austauschbar und können je nach Anwendungsgebiet gewählt werden. Da diese aber den zentralen Bestandteil der Transformation darstellen, müssen sie auch einige Bedingungen [Bän05] erfüllen:

1. Möglichst rasche und stabile Berechenbarkeit
2. Gute Lokalisierung im Zeitbereich
3. Gute Lokalisierung im Frequenzbereich
4. Orthonormale Basis

Bei den folgenden Erläuterungen wird auf mathematische Beweise gänzlich verzichtet. Als Verweis sei u.a. [LMR98] genannt, bei dem diese detailliert behandelt werden.

Die erste Forderung nach einer raschen und stabilen Berechenbarkeit mag auf den ersten Blick nicht unbedingt wichtig erscheinen, ist aber gerade in der Praxis nicht unerheblich. Dem *Fourier-Königreich*, wie es von Mallat [Mal99] benannt wurde, kann nur schwer etwas entgegengesetzt werden. Hohe Verbreitung, ausgereifte Algorithmen und weitreichende Hardware-Unterstützung der FFT erschweren es alternativen Transformationen, in der Praxis Fuß zu fassen. [Bän05]

Die zweite Bedingung, eine gute Lokalisierung im Zeitbereich, ist immer dann erfüllt, wenn die Grundfunktionen nur in einem begrenzten Zeitbereich einen deutlich von Null abweichenden Wert aufweisen. Analog dazu besagt die dritte Bedingung, dass deren Fourier-Transformierte ebenfalls eine begrenzte Ausdehnung aufweisen soll. Diese Be-

dingungen stehen in einem entscheidenden Zusammenhang zueinander: Eine im Frequenzbereich gut lokalisierte Funktion ist im Zeitbereich nur unscharf lokalisierbar, und umgekehrt. Das beste Beispiel hierfür ist die Fourier-Transformation, die durch eine ideale Frequenzlokalisierung bei gleichzeitig fehlender Zeitlokalisierung gekennzeichnet ist. Dieses Phänomen wird als *Unschärferelation* bezeichnet und zeigt, wenn auch nur im mathematischen Sinne, Ähnlichkeiten zur *Heisenberg'schen Unschärferelation* aus dem Bereich der Quantenmechanik. [Bän05]

Schließlich entspricht die vierte Bedingung der *orthonormalen* Basis der bereits in Abschnitt 2.2.5 ausführlich behandelten Orthogonalität, erweitert um den Begriff der *Norm*. Die Norm stellt in Funktionenräumen das Pendant zur Vektorlänge in der Vektorgeometrie dar. Wie in Definition 2.16 zu sehen, wird sie aus der Wurzel des Skalarprodukts berechnet [Bän05]:

Definition 2.16 (Norm)

$$\|f\| := \sqrt{< f, f >}\Big|_{f \in L^2(\mathbb{R})} = \sqrt{\int_{-\infty}^{\infty} |f(t)|^2 \, dt} \tag{2.19}$$

Diese Eigenschaft ermöglicht es, dass das ursprüngliche Signal direkt aus den analysierten Koeffizienten wiedergewonnen werden kann, ohne dass deren jeweilige Skalierung zuvor bei der Analyse oder später bei der Rückgewinnung angepasst werden muss. Beispiel 2.17 soll das verdeutlichen [Bän05]:

Beispiel 2.17 (Unterschied zwischen orthonormierten und orthogonalen Basen)
Will man eine Funktion mit Hilfe der Reihe

$$f(t) = \sum_k c_k \cdot \psi_k \tag{2.20}$$

entwickeln, so lassen sich die Koeffizienten c_k wie folgt berechnen:

$$c_k = < \psi_k, f > \tag{2.21}$$

Bilden die Funktionen von ψ allerdings nur eine orthogonale Basis, so müssen sie über folgenden Umweg berechnet werden:

$$c_k = \frac{< \psi_k, f >}{< \psi_k, \psi_k >} \tag{2.22}$$

Die Forderung nach einer zumindest orthogonalen Basis bewirkt eindeutige und redundanzfreie Koeffizienten, sie muss aber nicht zwangsläufig erfüllt sein. Betrachtet man

unter diesem Gesichtspunkt die Definition 2.15 der CWT, kann man erkennen, dass eine kontinuierliche, uneingeschränkte Skalierung und Verschiebung der Kernfunktionen zu keiner orthogonalen Basis führen kann, und somit schon ein konkretes Beispiel für die Verletzung dieser vierten Bedingung gefunden ist.

2.4.4 Konkrete Wavelet-Basisfunktionen

Welche Funktionen erfüllen die im vorangegangenen Abschnitt 2.4.3 aufgestellten Bedingungen? Welche Eigenschaften haben diese? In Abbildung 2-10 ist eine kleine Auswahl bekannter Wavelet-Funktionen und deren zugehörige Skalierungsfunktion dargestellt. Letztere wird im Zusammenhang mit der Multiskalenanalyse in Abschnitt 2.4.6 näher beschrieben.

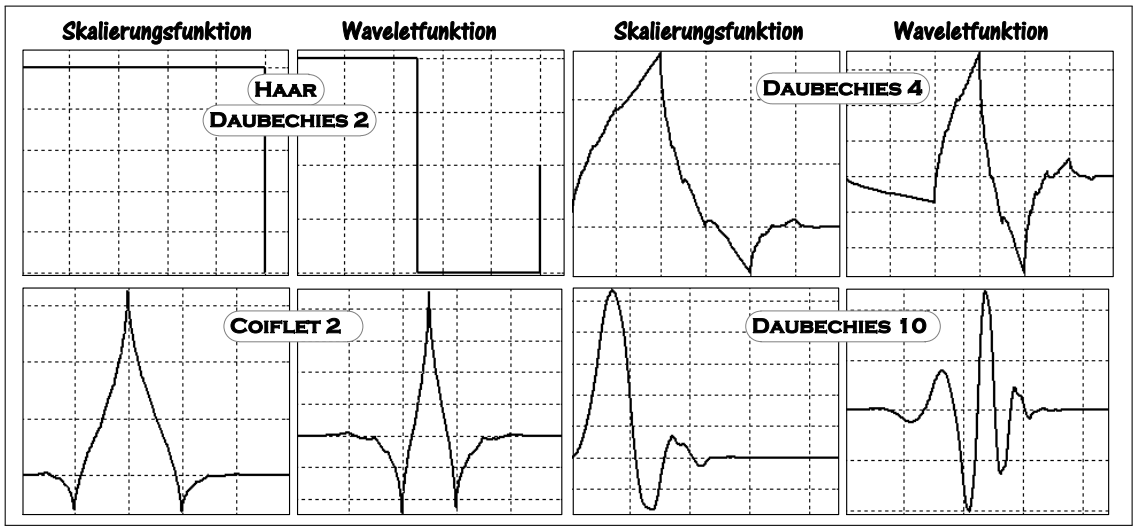

Abbildung 2-10: Verschiedene Wavelets und deren Skalierungsfunktionen (Matlab™)

Das *Haar-Wavelet* wurde 1910 von Alfred Haar definiert und wird damit als das älteste Wavelet angesehen. Es eignet sich aufgrund seiner Einfachheit gut zur Veranschaulichung der Wavelet-Transformation. Obwohl es eine gute Zeitlokalisation aufweist und eine orthonormale Basis damit gebildet werden kann, wird darauf in der Praxis aufgrund seiner Sprungstellen selten zurückgegriffen. Diese führen nämlich zu einer großen Ausdehnung und damit schlechten Lokalisierung im Frequenzbereich. [Bän05]

Ebenfalls als Pionierin im Bereich der Wavelet-Theorie gilt die belgische Mathematikerin Ingrid Daubechies. Die nach ihr benannten Daubechies-Wavelets weisen eine Besonderheit auf, die eine schnelle Berechnung in der digitalen Signalverarbeitung ermöglicht. Sie sind in ihrer Grundform nicht als Zeitfunktion, sondern in Form einer endlichen Anzahl an Filterkoeffizienten definiert. Diese Eigenschaft wird auch als *kompakter Träger* bezeichnet und ermöglicht in der praktischen Umsetzung den Einsatz sog. *FIR-Filter*, auf die in Abschnitt 2.4.8 eingegangen wird. Die Anzahl der Filterkoeffizienten wird zur

Unterscheidung im Namen des Wavelets mitangegeben, wie in Abbildung 2-10 zu sehen ist. Darin ist außerdem ein interessanter Zusammenhang zu sehen: Die erste Form der Daubechies-Wavelets, also jene mit nur zwei Filterkoeffizienten, ist identisch mit dem Haar-Wavelet. Damit wurde nachträglich nachgewiesen, dass auch das Haar-Wavelet in Form von Filterkoeffizienten dargestellt werden kann. [Bän05]

In manchen Anwendungsbereichen kann es vorkommen, dass noch weitere Eigenschaften von Wavelets gefordert werden. Ein Beispiel dafür ist die Symmetrie. Wie aus dem Bereich der digitalen Filter bekannt, kann mit einer endlichen, symmetrischen Impulsantwort ein linearer Phasengang und damit eine konstante Gruppenlaufzeit erzielt werden. [Ant05]

Eine vollständige Symmetrie ist bei orthogonalen Wavelets mit kompaktem Träger aber nicht möglich [Dau92]. Einzig das Haar-Wavelet würde beide Eigenschaften vereinen, erweist sich aber mangels Kontinuität bedingt durch die Sprungstellen in vielen Anwendungsfällen als untauglich [Bän05]. Eine Annäherung der Symmetrie ist allerdings realisierbar, wenn eine orthogonale Basis unverzichtbar ist. Hierfür wurden von Daubechies die Wavelets *Coiflet* und *Symmlet* entworfen [Cha04]. Anwendung finden sie z.B. bei der Kodierung von Bildern, bei der eine Verringerung der Phasenverschiebungen zur Verbesserung der Bildqualität beiträgt [Add02]. Umgekehrt kann auch Symmetrie als Ausgangspunkt festgelegt werden und z.B. auf vollständige Orthogonalität verzichtet werden. Ein Beispiel dafür sind sog. *biorthogonale Spline-Wavelets* [Bän05]. Für die Spektralanalyse spielt diese Eigenschaft aber nur eine untergeordnete Rolle, da in der Praxis die Phase nicht dargestellt wird und sich Fehler erst bei der erneuten Zusammensetzung des Signals im Zuge einer Synthese bemerkbar machen würden.

Die nachfolgende Tabelle 2-3 fasst abschließend die Eigenschaften einiger bekannter Wavelets zusammen:

Wavelet	**Kompakter Träger**	**Orthogonale Basis**	**Symmetrie**	**Weitere Eigenschaften**
Haar	ja	ja	ja	Keine Kontinuität
Daubechies	ja	ja	keine	schnelle Berechnung
Symmlet	ja	ja	annähernd	nahezu linearphasig
Coiflet	ja	ja	annähernd	nahezu linearphasig
Meyer	nein	ja	ja	diskrete Variante vorh.
Morlet	nein	nein	ja	nur CWT möglich
Mexikanischer Hut	nein	nein	ja	nur CWT möglich

Tabelle 2-3: Eigenschaften verschiedener Wavelets im Vergleich (Quelle:Matlab™)

2.4.5 Diskrete Wavelet-Transformation

Nach dem vorangegangenen Exkurs in das Gebiet der Wavelet-Basisfunktionen stellt sich nun die Frage, wie sich die Wavelet-Transformation diskretisieren lässt. In Anlehnung an die Diskretisierung der Fourier-Transformation in Abschnitt 2.2.7 wird auch hier das kontinuierliche Eingangssignal durch diskrete Abtastwerte ersetzt. Schwieriger wird es allerdings bei den Wavelets. Nicht nur die Wavelets selbst, sondern auch deren zweischichtige Abhängigkeit (Skalierung und Verschiebung) untereinander, müssen diskretisiert werden. Wie wirkt sich dieser Vorgang auf die Transformation aus? Kann im Zuge dessen das Problem der Redundanz beseitigt werden?

Alfred Haar hatte sich bereits 1910 festgelegte Skalierungs- und Verschiebungsschritte zu Nutze gemacht, um die Orthogonalität und damit Eindeutigkeit seiner Wavelet-Basis herzustellen. Definition 2.18 zeigt, welche konkreten Regeln für die Skalierung und Verschiebung der einzelnen Wavelets bezüglich dem als Vorlage dienenden Mother-Wavelet ψ festgelegt wurden. Die Variable m entspricht dabei der diskreten Skalierung, n der diskreten Verschiebung. Der vorangestellte Faktor dient zur Normalisierung der Wavelet-Basis [Bän05]. Die spezielle Form von Haar wurde später, wie in Gleichung 2.24 zu sehen, verallgemeinert. Diese Form sei nur der Vollständigkeit halber erwähnt, da sie in der Praxis eine eher untergeordnete Rolle spielt [Add02].

Definition 2.18 (Orthogonalität durch diskrete Skalierung und Verschiebung)

$$\psi_{m,n}(t) = 2^{-\frac{m}{2}} \cdot \psi\left(2^{-m} \cdot t - n\right) \quad m,n \in \mathbb{Z} \tag{2.23}$$

$$\psi_{m,n}(t) = \frac{1}{\sqrt{a_0^m}} \cdot \psi\left(\frac{t - n\, b_0\, a_0^m}{a_0^m}\right) \quad m,n \in \mathbb{Z} \tag{2.24}$$

Warum erfolgt hier die Skalierung in Form von Zweierpotenzen und nicht in linearen Schritten, wie z.B. bei dem Frequenzparameter der DFT? Dazu soll dazu zunächst Abbildung 2-11 angesehen werden. Sie zeigt auf der linken Seite, wie sich verschiedene diskrete Skalierungen und Verschiebungen nach Gleichung 2.23 auf das Haar-Wavelet auswirken. Anlehnend an Beispiel 2.8 aus Abschnitt 2.2.5, wird auch hier die Orthogonalitätsbedingung wie folgt angeschrieben:

$$< \psi_{m,n}, \psi_{m',n'} >= 0 \quad für\, m \neq m'\, oder\, n \neq n' \tag{2.25}$$

Grafisch kann die Orthogonalität durch Multiplikation der beiden zu vergleichenden Funktionen mit anschließender Berechnung der eingeschlossenen Fläche nachvollzogen werden, wie es im rechten Teil der Abbildung 2-11 gezeigt wird. Hält man sich an die Regel aus Gleichung 2.23 für die Ableitung der einzelnen Bestandteile der Haar-Wavelet-Basis, so ist die Aussage von Gleichung 2.25, ohne auf den mathematischen

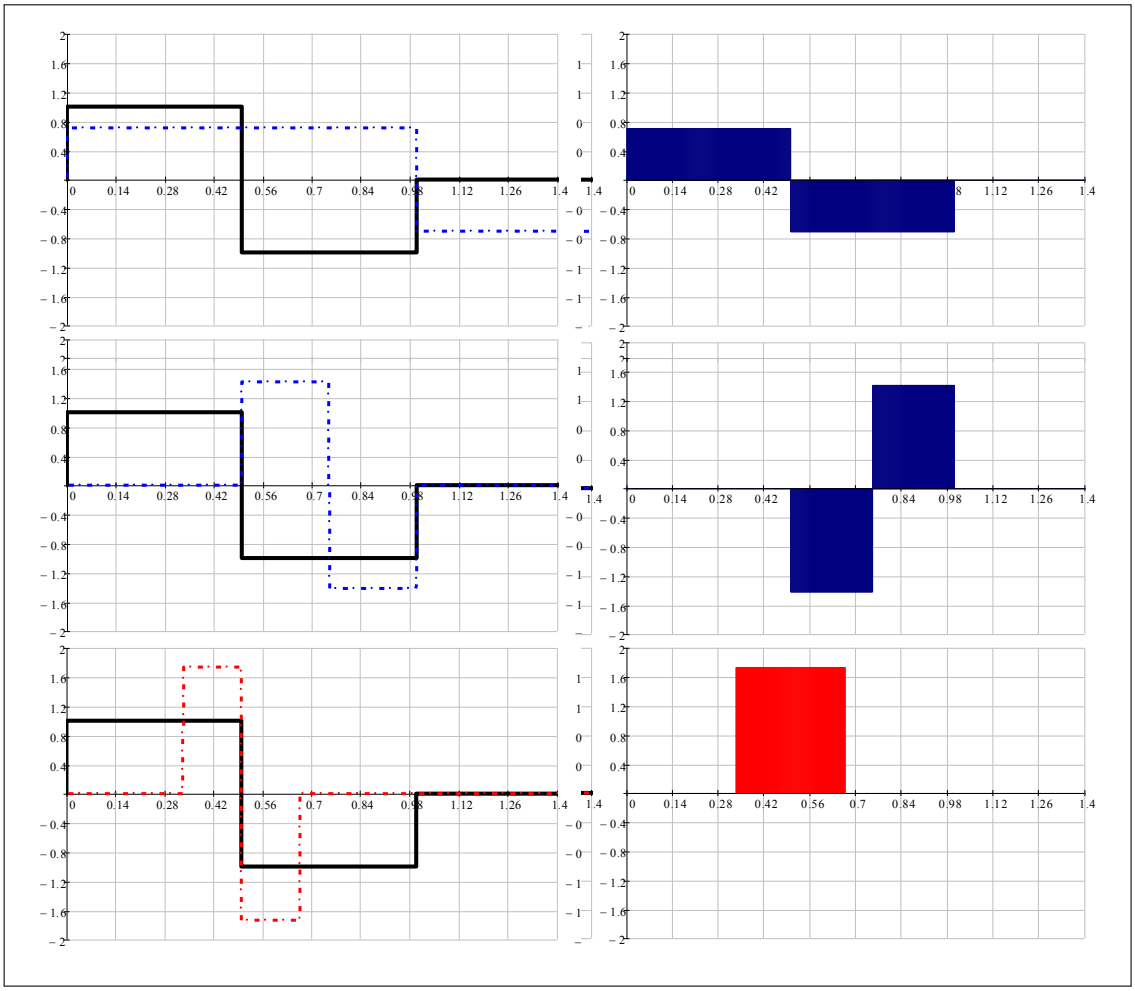

Abbildung 2-11: Orthogonalität bei Skalierung/Verschiebung des Haar-Wavelets (Mathcad^{TM})

Beweis einzugehen, immer zutreffend. Würde man aber eine lineare Skalierung einsetzen, also z.B. $m = 3$, wie in der letzten Zeile der Abbildung 2-11, dann ist die Orthogonalität nicht erfüllt, da das Skalarprodukt nicht Null ergibt.

Aus diesem Grund muss, wenn Orthogonalität gewünscht wird, das oben definierte logarithmische Raster eingehalten werden. Die Basis 2 wird dabei in der Praxis besonders häufig aufgrund ihrer in der Digitaltechnik effektiven Berechenbarkeit verwendet und wird auch als *dyadisches* Raster bezeichnet [Add02]. Zur Veranschaulichung ist es in Abbildung 2-12 grafisch dargestellt. Im Gegensatz zum linearen Raster der FFT ist hierin die gute Frequenzauflösung bei tiefen Frequenzen und die gute Zeitauflösung bei hohen Frequenzen erkennbar. Zudem ist die immer geltende Unschärferelation zwischen Frequenz- und Zeitauflösung deutlich sichtbar.

Die diskrete Wavelet-Transformation - kurz DWT - und ihre inverse Form sind in Definition 2.19 unter Verwendung des Zusammenhangs der Gleichung 2.23 bzw. 2.24 angeschrieben. [Add02]

Definition 2.19 (Diskrete Wavelet-Transformation(2.26) und Inverse DWT(2.27))

$$F_{m,n} = \int_{-\infty}^{\infty} x(t) \cdot \psi_{m,n}(t)\, dt \tag{2.26}$$

$$x(t) = \sum_{m=-\infty}^{\infty} \sum_{n=-\infty}^{\infty} F_{m,n} \cdot \psi_{m,n}(t) \tag{2.27}$$

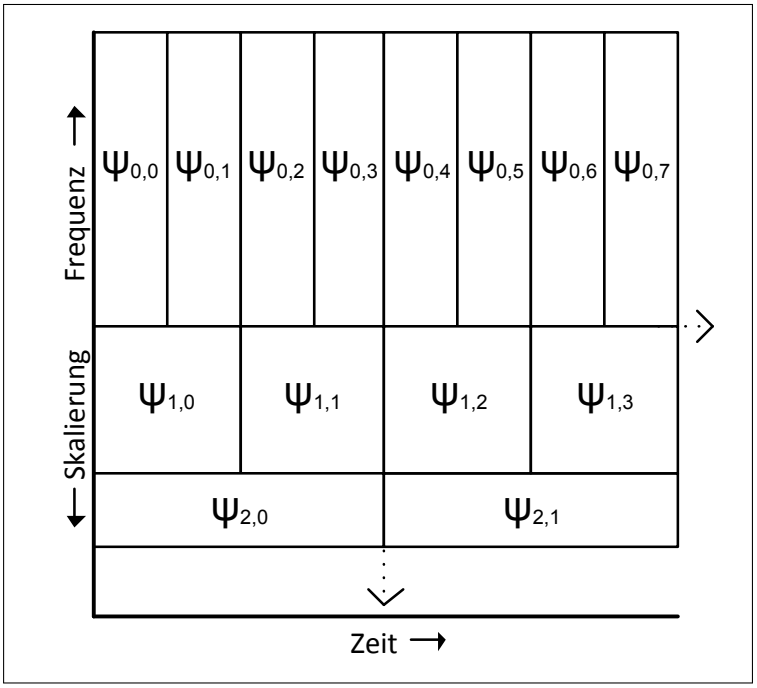

Abbildung 2-12: Dyadisches Raster der diskreten Wavelet-Transformation

Es mag zunächst verwirrend erscheinen, dass die Definition, entgegen ihrer Namensgebung, nicht in vollständig diskreter Form z.B. für die Berechnung mit Abtastwerten vorliegt, sondern in Form einer Wavelet-Reihe mit diskreten Beziehungen der Wavelets zueinander. Nichtsdestotrotz ist aber gerade durch die diskreten Skalierungs- und Verschiebungsschritte das kontinuierliche Integral nur mehr in jenem diskreten Raster bestimmt. Das wiederum führt zu diskreten Koeffizienten und ermöglicht die Rückgewinnung des ursprünglichen Signals durch einfache Summierung. [Add02]

2.4.6 Multiskalenanalyse

In diesem Abschnitt wird der letzte Schritt zwischen der theoretischen DWT aus dem vorangegangenen Abschnitt 2.4.5 und einem in der Praxis umsetzbaren Algorithmus zur Berechnung der schnellen Wavelet-Transformation vollzogen. Dazu greift man auf bisher Bewährtes zurück und versucht auch hier einen rekursiven Algorithmus zu finden, der sich Synergien zu Nutze macht.

Die dabei verfolgte Grundidee ist, den Skalierungsfaktor auf einen konstanten Wert *einzufrieren* und dadurch zum einen zu untersuchen, wie die Ausgangsfunktion durch eine Summe von Grundbausteinen angenähert werden kann und zum anderen herauszufinden, welche Verbindung zur nächst höheren bzw. niedrigeren *Skalierungsstufe* besteht. Ziel dabei ist es, eine Brücke zwischen der Berechnung der Koeffizienten $< f, \psi_{m,n} >$ und $< f, \psi_{m+1,n} >$ zu schaffen, über welche die Berechnung - im Speziellen des Integrals - nicht für jede Skalierungsstufe von Neuem begonnen werden muss. [Bla03]

In der Multiskalenanalyse behilft man sich dabei der sog. *Skalierungsfunktion*, die als Partner der Wavelet-Funktion angesehen werden kann. Die Ausgangsfunktion wird in Analogie zum Mother-Wavelet auch als *father scaling function* bezeichnet. Sie wurde bereits in Abbildung 2-10 vorweggenommen und für die dort gezeigten Wavelets dargestellt. Die Skalierungsfunktionen stehen zueinander in der gleichen Relation wie die Wavelets [Add02]:

$$\phi_{m,n}(t) = 2^{-\frac{m}{2}} \cdot \phi\left(2^{-m} \cdot t - n\right) \qquad m,n \in \mathbb{Z} \tag{2.28}$$

Die wichtigsten Eigenschaften der Skalierungsfunktionen sind zum einen die Orthogonalität bezogen auf eine fixierte Skalierungsstufe und zum anderen deren Integral bzw. Energie, die im Gegensatz zu den Wavelet-Funktionen nicht 0, sondern 1 ist:

$$\int_{-\infty}^{\infty} \phi(t)\, dt = 1 \tag{2.29}$$

Das kann auch in Abbildung 2-10 erkannt werden, bei der die Skalierungsfunktionen überwiegend Werte auf der positiven Y-Achse aufweisen [Add02]. Diese Eigenschaft wird als *Mittelungseigenschaft* [Bän05] bezeichnet. Verwendet man nämlich eine Linearkombination von Skalierungsfunktionen anstelle der Wavelet-Funktionen, um eine Ausgangsfunktion anzunähern, so erhält man eine gemittelte Approximation dieser Ausgangsfunktion. Dies ist deshalb der Fall, weil sich nun die Mittelungseigenschaft der als Kern dienenden Skalierungsfunktionen auf die Approximation überträgt. [Add02]

Unter Ausnutzung aller Eigenschaften der Skalierungsfunktionen gelingt es auch, die Wavelet-Funktion als Linearkombination von Skalierungsfunktionen der nächst kleineren Skalierungsstufe anzuschreiben [Bän05]:

$$\psi = \sum_{k \in \mathbb{Z}} g_k \cdot \phi_{-1,k} \tag{2.30}$$

Die in Gleichung 2.30 dargestellte Linearkombination entspricht der zuvor gesuchten Brücke zwischen den Skalierungsstufen und stellt zudem die Verbindung zwischen den Skalierungs- und den Wavelet-Funktionen her.

Klammert man mathematische Herleitungen und Beweise aus, so kann man zu folgen-

der, vereinfachter Sichtweise der MSA kommen und damit den Algorithmus der schnellen Wavelet-Transformation - kurz FWT - erahnen:

Folgerung 2.20 (Vereinfachte Beschreibung der Multiskalenanalyse)
In der Multiskalenanalyse wird die Wavelet-Funktion für die Analyse der Details des Signals, die Skalierungsfunktion dagegen für den Übergang in die nächst gröbere Stufe - also vereinfacht zum „Hinaus-zoomen“ - eingesetzt. Das bewirkt, dass bereits Analysiertes ausgeblendet wird, was zu einer effizienteren Berechnung beiträgt. Diese nächste Vergröberungsstufe wird wiederum in gleicher Form analysiert, bis die gewünschte Genauigkeit erreicht ist.

2.4.7 DWT und FWT im Frequenzbereich

Um den nicht so einfach vorstellbaren Vorgängen der Wavelet-Transformation besser folgen zu können, hilft eine Betrachtung im Frequenzbereich. Einen guten Ausgangspunkt hierfür stellen die Wavelets dar. Wie sehen diese im Frequenzbereich aus? In Abschnitt 2.4.3 wurden Bedingungen formuliert, die Wavelets erfüllen müssen bzw. deren Einhaltung zumindest erwünscht ist. Eine von ihnen war die Forderung nach einer guten Lokalisierung im Frequenzbereich. Diese ist dann erfüllt, wenn die Fourier-Transformierte des Wavelets ein deutlich bandbegrenztes Verhalten zeigt.

Die gesamte Wavelet-Basis wird aus verschobenen und skalierten Versionen des Mother-Wavelets gebildet. Wie wirken sich diese Operationen auf den Frequenzbereich aus? Eine Verschiebung im Zeitbereich hat keinerlei Einfluss auf das Betragsspektrum, sondern nur auf das hier nicht behandelte Phasenspektrum. Eine Streckung im Zeitbereich dagegen führt zu einer Stauchung des Spektrums. Außerdem bewirkt eine Streckung eine Veränderung der Grundfrequenz, die dadurch tiefer wird. Die Auswirkungen auf das bandbegrenzte Wavelet sind in Abbildung 2-13 skizziert. [Val04]

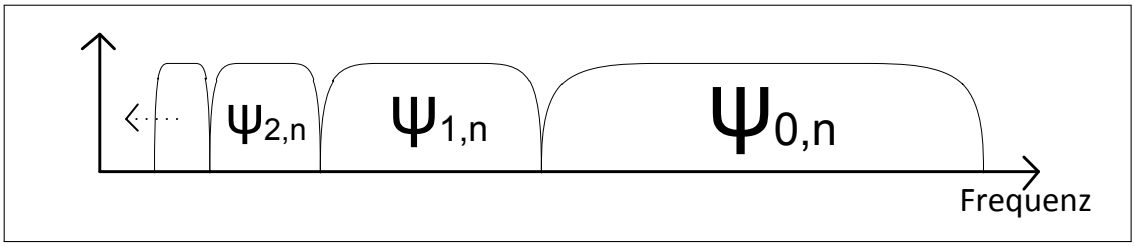

Abbildung 2-13: Diskrete Wavelet-Transformation im Frequenzbereich

Daraus lässt sich auch deutlich der Nachteil der DWT erkennen: Jede Streckung bewirkt eine erweiterte Frequenzabdeckung hinsichtlich der tiefen Frequenzen. Eine vollständige Abbildung ist allerdings nur mit unendlich vielen Koeffizienten und Approximationsschritten möglich. Bei aufmerksamer Betrachtung fällt zudem auf, dass alle Amplituden

mit gleicher Höhe dargestellt wurden, obwohl eine reine Streckung des Signals im Zeitbereich auch eine Veränderung der Amplitude im Frequenzbereich zur Folge hätte. Die Erklärung dafür ist die normalisierte Basis der Wavelets, die diese Unterschiede ausgleicht. Schließlich lässt sich aus Abbildung 2-13 auch erkennen, wie eine solche DWT praktisch umgesetzt werden könnte: Mit Hilfe einer parallelen Anordnung von Bandpassfiltern, deren Bandbreite in konstanter Relation zu deren Mittenfrequenz steht, also einer sog. *Filterbank* konstanter Güte (*constant Q*). Der Ausgang der einzelnen Filter entspricht dann den Koeffizienten der jeweiligen Skalierungsstufe.

Wie ist die MSA im Frequenzbereich zu deuten und wie lässt sich die daraus resultierende schnelle Wavelet-Transformation beschreiben? Die Antwort darauf ist in Abbildung 2-13 grafisch dargestellt.

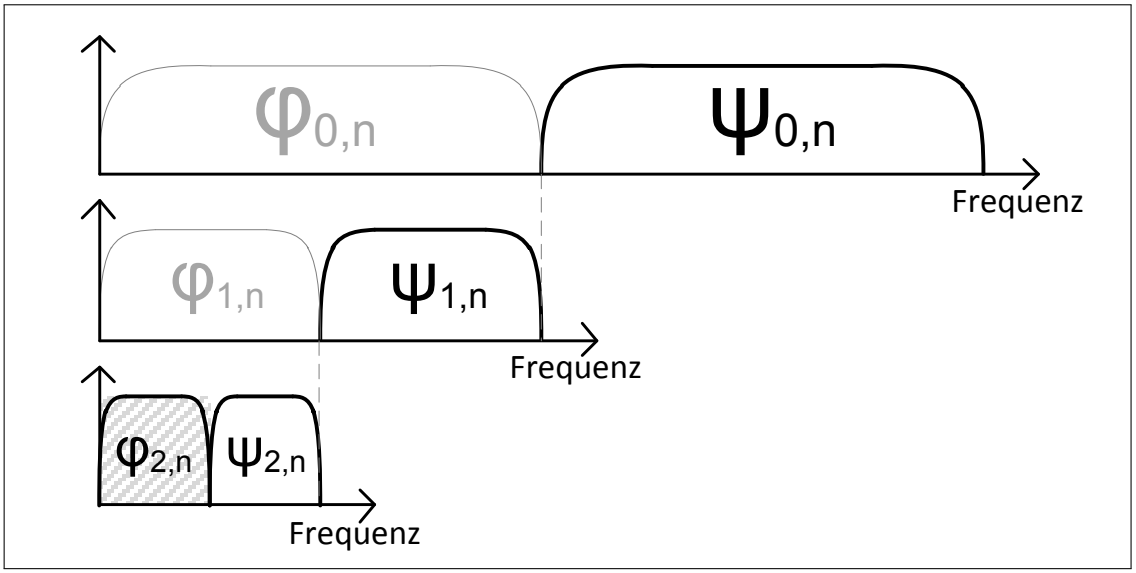

Abbildung 2-14: Schnelle Wavelet-Transformation im Frequenzbereich

Die mittelnde Eigenschaft der Skalierungsfunktion macht sich im Frequenzbereich als Tiefpass bemerkbar. Sie blendet die bereits ermittelten Details vor dem Übergang in die nächste Skalierungsstufe aus und wirkt zusätzlich - wie von Valens passend bezeichnet - als *Korken*, der die letzte Lücke im unteren Frequenzbereich schließt und somit auch nach endlich vielen Koeffizienten eine vollständige Abbildung ermöglicht. [Val04]

2.4.8 Praktische Umsetzung der FWT als Filterbank

Mit den Erkenntnissen aus den vorangegangenen Abschnitten steht einer praktischen Umsetzung der schnellen Wavelet-Transformation nichts mehr im Wege. Das Blockschaltbild für die Analyse und für die rekonstruierende Synthese ist in Abbildung 2-15

dargestellt [Bän05].

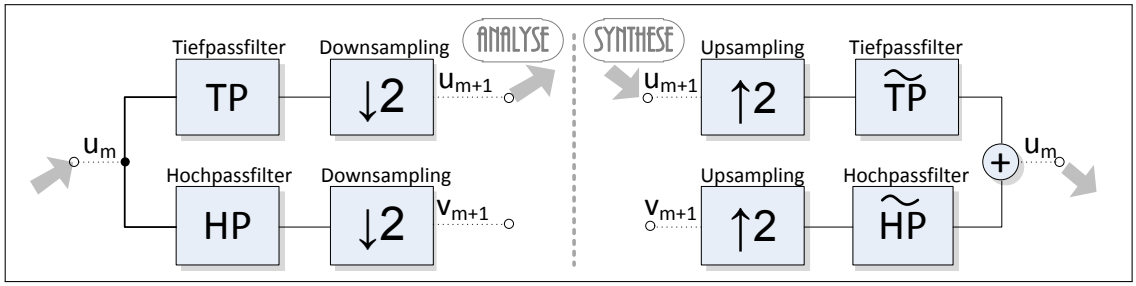

Abbildung 2-15: FWT Filterbank

Abbildung 2-15 zeigt, dass lediglich Standardelemente wie Filter, Up- bzw. Downsampler und Summierer für die Realisierung notwendig sind. Damit ist die FWT bzw. deren Umkehrung IFWT signifikant einfacher und effektiver implementierbar als z.B. die DWT. Zu beachten ist, dass hier nur jeweils eine Stufe des rekursiven Algorithmus abgebildet ist. Der Gesamtaufbau mit mehreren Filterstufen wird unter dem Begriff *Filterbank* zusammengefasst. Die schattierten Pfeile bei u_m und u_{m+1} zeigen jene Stellen, an denen die einzelnen Filterstufen zusammengeschaltet werden. Im Beispiel der Analyse heißt das, dass u_m mit dem Ausgang der vorangegangenen Stufe und u_{m+1} mit dem Eingang der nächsten Stufe zu verbinden ist. Die v_m entsprechen dabei den Wavelet-Koeffizienten, wobei m der Skalierungsfaktor ist, der mit jeder Stufe steigt und somit zur Analyse des nächst tieferen Frequenzbereichs führt. Der letzte Ausgang u_{m+n}, an den keine weitere Stufe mehr angrenzt, liefert die abschließenden Koeffizienten der Skalierungsfunktion, welche wie in Abschnitt 2.4.7 beschrieben, die Lücke im niederfrequenten Bereich schließen. [Bän05]

Als Filter kommen standardmäßig FIR-Filter zum Einsatz, da die in der Praxis gebräuchlichsten Wavelets die vorteilhafte Eigenschaft eines kompakten Träger aufweisen. Näheres dazu kann in Abschnitt 2.4.4 nachgelesen werden.

Das *Downsampling* erfolgt durch Entfernen jedes zweiten Abtastwerts, wobei im Design auf Aliasing Rücksicht genommen werden muss [Mer99]. Eine wirkungsvolle Maßnahme dazu wird später in Abschnitt 2.4.10 beschrieben. Mit jeder Stufe wird also das betrachtete Frequenzband in ein oberes und unteres Frequenzband aufgeteilt, wobei nur das Untere in der nächsten Stufe weiterverarbeitet wird. Dieses Verfahren wird auch als *Subband Coding* bezeichnet.

Analog zur FFT werden auch hier aus praktischen Gründen immer $N = 2^k$ mit $k \in \mathbb{N}$ Abtastwerte in Blöcken verarbeitet, damit das wiederholte Halbieren von Abtastwerten ohne Rest nach k Schritten endet. Durch das Weglassen jedes zweiten Abtastwerts halbiert sich mit jeder Stufe die Zeitauflösung, gleichzeitig verdoppelt sich aber die Fre-

quenzauflösung, da sich die zu analysierende Bandbreite halbiert. Das deckt sich mit dem in Abschnitt 2.4.1 beschriebenen, dyadischen Raster [Bän05].
Dieser Aspekt darf speziell dann nicht vergessen werden, wenn man die Koeffizienten zur Darstellung eines Spektrums heranzieht. Im Gegensatz zu den frequenzlinearen Koeffizienten der FFT, die äquidistant dargestellt werden können, muss hier eine Umrechnung zwischen Skala und Frequenz, sowie eine Beachtung der unterschiedlichen Zeitauflösungen erfolgen.

Die Synthese erfolgt umgekehrt zur Analyse. Es werden die gleichen Filter verwendet, allerdings mit inversen, im Blockschaltbild durch ein „$\sim$“ gekennzeichneten Filterkoeffizienten. Das Upsampling erfolgt durch abwechselndes Einfügen von Abtastwerten mit dem Wert Null. Es wird dadurch kein Fehler verursacht, wie man anfänglich vermuten könnte, da die nachfolgenden Filter und die Summierung das Ihre dazu beitragen, dass das Originalsignal - wiederum ohne auf den mathematischen Beweis einzugehen - perfekt rekonstruiert werden kann. Filterbänke dieser Art werden daher auch als *perfect reconstruction* - kurz *PR* - klassifiziert. [Bän05]

Abschließend sei noch erwähnt, dass es aus theoretischer Sicht falsch wäre, die zu analysierenden Abtastwerte direkt dem Eingang der ersten Filterbankstufe zuzuführen, da dieser ausschließlich für den Ausgang einer vorangegangenen Stufe vorgesehen ist. Wie bei allen Verfahren, in denen speichernde Elemente vorkommen, existiert auch hier eine Anfangswertproblematik. Der durch direkte Übernahme der Abtastwerte entstehende Fehler wird aber in der Praxis vernachlässigt, da er sich aufgrund der positiven Eigenschaften der Wavelets nur lokal auswirkt. [Bän05]

2.4.9 Wavelet Pakete

Im vorangegangenen Abschnitt 2.4.8 wurde gezeigt, wie die FWT als mehrstufige Filterbank implementiert wird, und dadurch eine Analyse nach dem festgelegten dyadischen Raster umgesetzt werden kann. Nähert man sich weiter der praktischen Signalverarbeitung an, so stellt sich die Frage, ob hier nicht auch der Hochpasszweig weiterverarbeitet werden könnte. Damit würde man vom vorgegebenen Raster abweichen, das bewusst, wie in Abschnitt 2.4.5 beschrieben, zur Sicherstellung der Orthogonalität definiert wurde. Muss man, wenn man diesen Weg einschlägt, auf diese sehr nützliche Eigenschaft verzichten? Oder lässt sich Orthogonalität auch auf anderem Wege erzielen?

Coifman, Meyer und Wickerhauser haben dieses Thema untersucht und dabei die *Wavelet Pakete* definiert [Rus92]. Erweitert man die Filterbank aus Abbildung 2-15 so, dass auch jeder Hochpasszweig weiter unterteilt wird, so ähnelt die daraus resultierende Struktur einem binären Baum, wie er in Abbildung 2-16 dargestellt ist [Bän05]. Gegen-

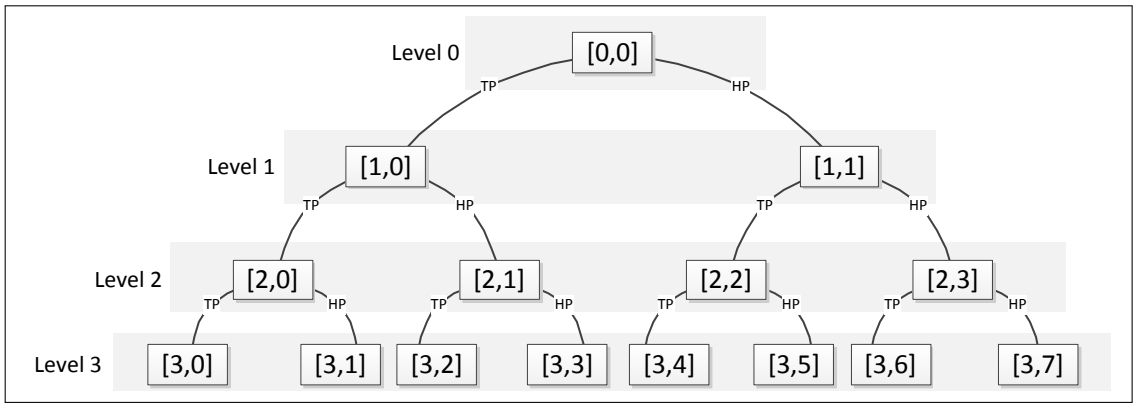

Abbildung 2-16: Baumstruktur der Wavelet-Pakete

über einer FWT-Filterbank ergeben sich somit wesentlich mehr Knoten, die als Wavelet-Pakete bezeichnet werden. Wenn man sie alle als Koeffizientenquelle einsetzen würde, führt das zu einer deutlich ausgeprägten Redundanz. Die zentrale Frage an dieser Stelle ist, wie durch geschickte Wahl der Knoten eine redundanzfreie und somit orthogonale Analyse ermöglicht werden kann.

Wie kann Orthogonalität in der Darstellung des Frequenz/Zeit-Rasters der Filterbank beschrieben werden? Die Amplitudengänge der eingesetzten Filter müssen einerseits das gesamte zu analysierende Frequenzspektrum lückenlos abdecken, dürfen sich aber andererseits nicht überschneiden [Bän05]. In den Abbildungen 2-17 und 2-18 werden verschiedene Möglichkeiten der Knotenwahl und deren Auswirkungen auf das Frequenz/Zeit-Raster gezeigt. Die daraus resultierende Transformation wird als *Wavelet Paket Transformation* - kurz WPT - bezeichnet.

Abbildung 2-17 zeigt im oberen Teil, dass die FWT als ein Spezialfall der Wavelet-Paketbasis aufgefasst werden kann, bei dem die Knoten des rechten Astes, wie dargestellt, gewählt werden. Für eine optimale Frequenzauflösung könnten die Knoten der untersten Stufe gewählt werden, wie im unteren Teil der Abbildung 2-17 zu sehen ist. In der zugehörigen Rastertabelle fallen dabei zwei Punkte auf: Zum einen ist zu erkennen, dass auch hier der Grundsatz der Unschärferelation ausnahmslos gilt, da die gute Frequenzauflösung zu Lasten der Zeitauflösung geht. Zum anderen fällt auf, dass die Reihenfolge der nach Frequenz absteigend sortierten Knoten im Frequenz/Zeit-Raster durcheinander gebracht scheint. Tatsächlich ist diese Umordnung aber bewusst vorgenommen worden, da die Reihenfolge der Knoten nicht mit jener der zugehörigen Frequenzen übereinstimmt.

Werden alle Knoten aus einer Stufe entnommen, wie es hier der Fall ist, so kann die Umschlüsselung der Indizes über den *Grey Code* erfolgen [Mal99]. Aus den ursprünglichen Paket-Indizes einer Stufe, also z.B. $0,1,2,3,4,5,6,7$, werden somit die als Dezimalzahlen dargestellten Grey-Code Symbole $0,1,3,2,6,7,5,4$. Ein stufenunabhängiger

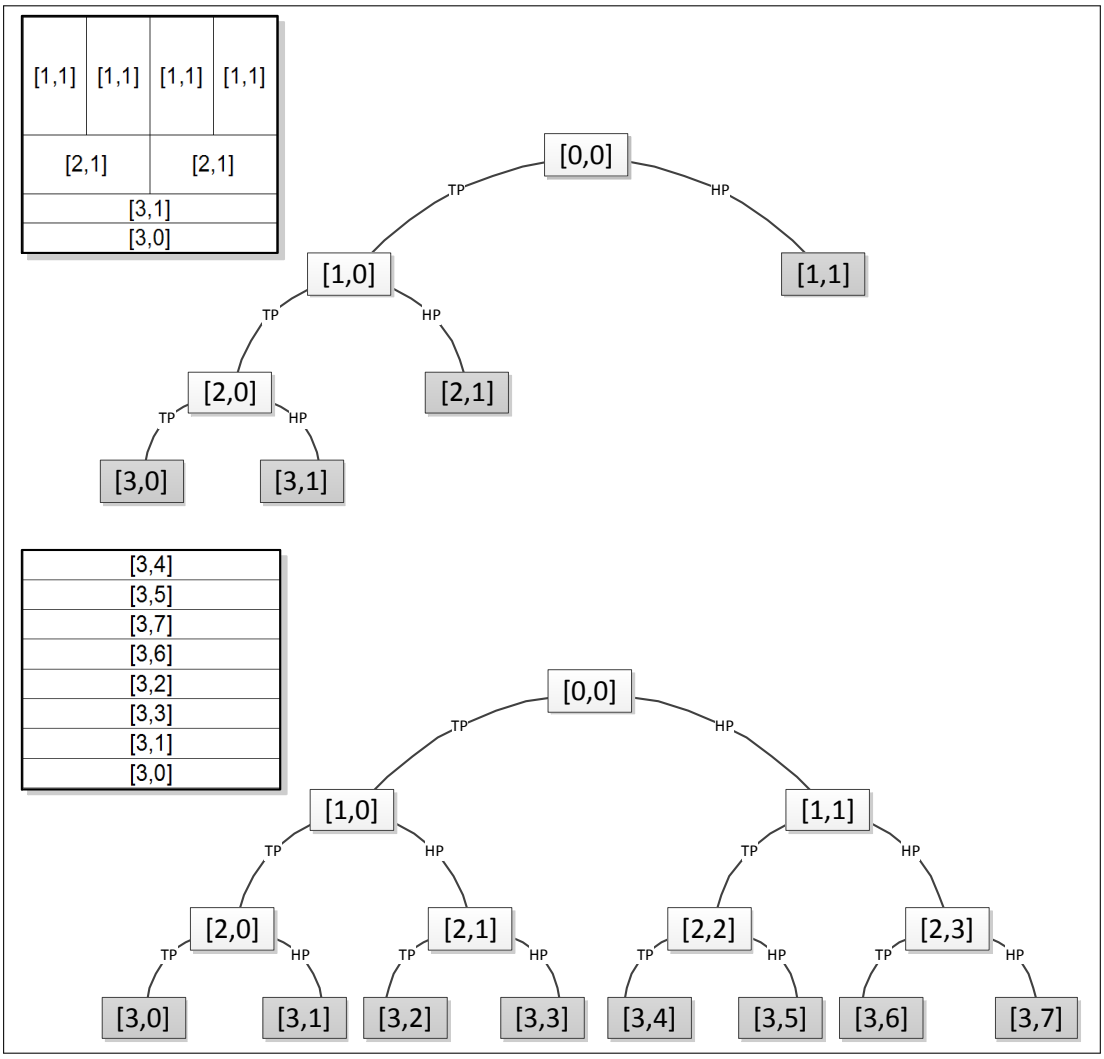

Abbildung 2-17: Wavelet-Paketbasis der DWT(oben) und frequenzauflösungs-optimiert(unten)

Algorithmus wird dazu noch in Abschnitt 7.2 vorgestellt, welcher nach der Beschreibung von [Kap02] implementiert wurde. Näheres kann zudem in [JCH01] nachgelesen werden.

Abbildung 2-18 zeigt im oberen Teil, wie flexibel die Einteilung des Frequenz/Zeit-Rasters gestaltet werden kann. Die Wavelet-Pakete eröffnen damit gänzlich neue Möglichkeiten. Die Knotenwahl, die bisher manuell vorgenommen wurde, kann auch adaptiv, also automatisch an das Signal angepasst, erfolgen. Das bedeutet, dass die Knoten mit einer sog. *cost function* bewertet und zur Laufzeit ausgewählt werden. Somit lässt sich für jeden Verarbeitungsblock die für das Signal nach bestimmten Kriterien *beste Basis* finden. Weist das Signal z.B. kurzfristig viele schnell aufeinanderfolgende Frequenzänderungen auf, so könnte automatisch zu einer höheren Zeitauflösung und anschließend wieder zurück gewechselt werden [Mal99]. Anwendung finden diese Algorithmen z.B. in der Rauschunterdrückung, bei der gezielt jene Koeffizienten auf Null gesetzt werden, die das Störsignal repräsentieren [FK01].

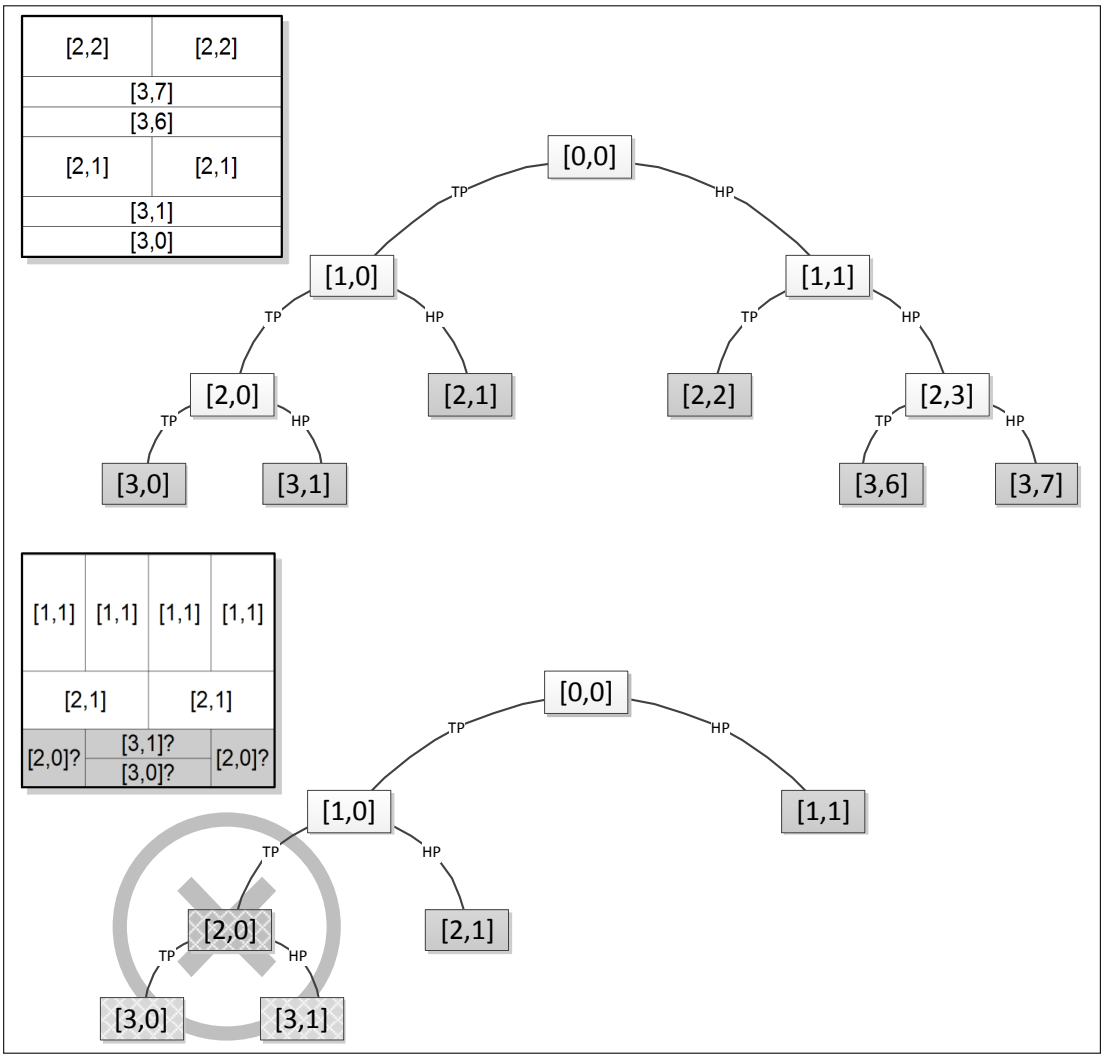

Abbildung 2-18: Individuelle(oben) und redundante(unten) Wavelet-Paketbasis

Schließlich zeigt Abbildung 2-18 unten noch eine bezüglich Orthogonalität inkorrekte Knotenwahl, die zu einer redundanten Wavelet-Paketbasis führen würde.

2.4.10 Ausblick Wavelets

In diesem Abschnitt soll abschließend ein kurzer Ausblick über weiterführende Themen im Bereich der Umsetzung der Wavelet-Transformation gegeben werden. Bei Interesse kann in der jeweils angegebenen Literatur Näheres nachgelesen werden.

Trotz der bereits schnellen Implementierung in Form einer Filterbank können noch weitere effizienzsteigernde Maßnahmen getroffen werden. Man könnte sich z.B. fragen, ob das in Abbildung 2-15 dargestellte Downsampling nach dem Filter wirklich effizient platziert ist. Muss das Filter wirklich alle Ausgabewerte berechnen, wenn doch nur

die Hälfte davon verwendet werden? Die Antwort darauf liefert die sog. *Polyphasen*-Filterstruktur, bei der das Downsampling vor das Filter verlagert wird, was die zu verarbeitenden Abtastwerte halbiert. Einhergehend damit muss die Filterstruktur modifiziert werden, indem die geraden und ungeraden Koeffizienten getrennt und die ungeraden Koeffizienten zusätzlich noch einer Verzögerung zugeführt werden. Eine sehr anschauliche Beschreibung dazu kann im Artikel [Val99] gefunden werden. Eine detailliertere Betrachtung, insbesondere in Bezug auf andere Downsampling-Raten, kann in [PM96] nachgelesen werden.

Die Polyphasen-Filterstruktur wird in der Praxis häufig in Kombination mit dem sog. *Quadratur Spiegel Filter* bzw. *quadratur mirror filter* - kurz QMF - realisiert. Voraussetzung dafür ist, dass die Frequenzgänge des Hoch- und Tiefpassfilters zueinander spiegelsymmetrisch sind, d.h. dass deren Ausgänge in Summe, ungeachtet der Phasenverschiebungen, wieder das Originalsignal und damit einen konstanten Frequenzgang ergeben. Diese Voraussetzung wird durch die Bedingungen an die Skalierungsfunktionen im Zuge der Multiskalenanalyse erfüllt, wie in [KWW] gezeigt wird. QMF weisen eine in der Praxis besonders vorteilhafte Eigenschaft auf: Die Strukur unterdrückt Aliasing vollständig, da sich die entgegengesetzten Anteile aus dem Hoch- und Tiefpasszweig auslöschen. Näheres dazu kann u.a. in [Mer99] nachgelesen werden.

Ein anderer, in der Datenkompression immer häufiger anzutreffender Ansatz zur Umsetzung der Wavelet-Transformation, aber auch zum Finden neuer Wavelet-Basen, wird mit dem sog. *Lifting*-Schema verfolgt. Dabei wird die Aufteilung in Details und Vergröberung nicht durch Filter im herkömmlichen Sinne, sondern durch Vorhersage und Korrektur (*prediction and update*) erreicht. Abbildung 2-19 zeigt das Prinzip in Form eines Blockschaltbildes.

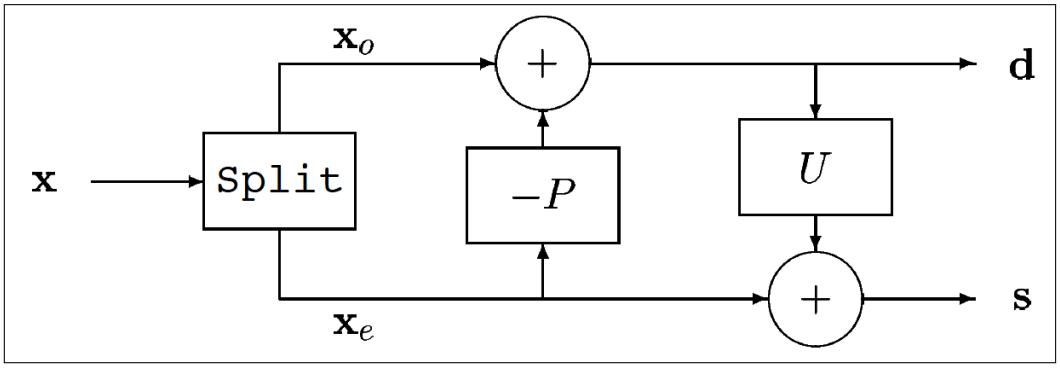

Abbildung 2-19: Lifting-Schema (Quelle: [DS98])

Die Vorhersage wird durch Interpolation der geradzahligen Abtastwerte realisiert, wobei eine möglichst genaue Annäherung der ungeradzahligen Abtastwerte das Ziel ist. Die Differenz zwischen den vorhergesagten und den tatsächlichen Abtastwerten wird

als Detail ausgeführt und entspricht, in Analogie zur DWT, den Wavelet-Koeffizienten [DS98].

Der Korrekturschritt ist notwendig, um den Signaldurchschnitt aufrecht zu halten und entspricht somit, in Analogie zur DWT, den Koeffizienten der Skalierungsfunktion. Durch Umformung können bestehende, als Filterkoeffizienten gegebene Wavelets in das Lifting-Schema übertragen werden. Darüber hinaus können neue Wavelet-Basen intuitiv über die Sichtweise der Interpolation entworfen werden. [DS98]

Für eine eingehendere Betrachtung sind die Artikel [DS98] und [Swe97] zu empfehlen. Verschiedene Implementierungsvarianten können u.a. in [Mal99] nachgeschlagen werden.

2.5 Wavelet- und Fourier-Transformation im Vergleich

Um das Kapitel der Spektralanalysegrundlagen abzuschließen, ist in Tabelle 2-4 kurzer Vergleich der Eigenschaften der Wavelet- und Fourier-Transformationen dargestellt:

Eigenschaft	**Fourier-Transformation**	**Wavelet-Transformation**
Frequenzauflösung	ideal	entwurfsabhängig
Zeitauflösung	gering, gefenstert	entwurfsabhängig
Frequenzeinteilung	linear	logarithmisch,linear, oder individuell
Zeiteinteilung	linear	logarithmisch,linear, oder individuell
Adaptive Auflösung	nicht möglich	mit Wavelet-Paketen möglich
Performance	sehr hoch	hoch
Verbreitung	sehr hoch	mittel
Anwendungen	Analyse, Kommunikationstechnik, etc.	Analyse, Synthese, Kompression Störunterdrückung, etc.

Tabelle 2-4: Fourier- und Wavelet-Transformation im Vergleich

Obwohl der Fourier-Transformation in der Praxis begründet durch eine fallweise schnellere Implementierung und bessere Hardwareunterstützung noch oft der Vorzug gegeben wird, wie es z.B. die Entscheidung für OFDM und gegen DWMT im Kommunikationsbereich gezeigt hat [Bän05], eröffnet die Wavelet-Transformation durch ihre Eigenschaften und Flexibilität vielerlei Möglichkeiten. Diese finden insbesondere im Bereich der Kompression, Rauschunterdrückung, Bildbearbeitung und Mustererkennung Anwendung.

3 Grundlagen der Psychoakustik

3.1 Einleitung

Dieses Kapitel gibt einen kurzen Einblick in jene Bereiche der Psychoakustik, die für die Bewertung eines auf technischem Wege ermittelten Spektrums von Bedeutung sind. Die psychoakustischen Aspekte werden dabei hinsichtlich ihrer Auswirkungen und Darstellbarkeit, wie auch in Bezug auf die Eigenschaften der Fourier- und Wavelet-Transformation, untersucht.

3.2 Hörsinn

Der menschliche Hörsinn, allen voran das Gehirn in Zusammenarbeit mit dem Gehör, nimmt Schallereignisse auf wesentlich komplexere Art wahr, als dies mit Mikrofonen und anderen technischen Mitteln denkbar wäre.

Die Schallwellen werden vom Außenohr aufgenommen, treffen dort auf das Trommelfell und werden von der Gehörknöchelchen-Kette *Hammer*, *Amboss* und *Steigbügel* für den energiereicheren Flüssigkeitsschall - vergleichbar mit einer Impedanzwandlung - aufbereitet. Die im Innenohr befindliche *Basilarmembran* wird durch die Schallwellen, die sich dort in Form von Wanderwellen in der *Endolymphflüssigkeit* ausbreiten, ausgelenkt und krümmt damit die dort befindlichen inneren *Haarzellen*. Diese wiederum leiten über die Hörnerven die Signale an das Gehirn weiter. Jede der ca. 3500 inneren Haarzellen ist dabei einem bestimmten Frequenzbereich zugeordnet, der sich aufgrund der topologischen Lage der Haarzelle ergibt. Hohe Frequenzen werden nahe dem Steigbügel, tiefe Frequenzen dagegen nahe der Schneckenspitze aufgenommen. Diese Frequenztrennung lässt sich bis in die Hörrinde des Großhirns nachweisen. Im Gehirn werden die Schallinformationen parallel und hinsichtlich mehrerer Aspekte wie Laufzeitdifferenzen, Muster, etc. verarbeitet. Aus den vergleichsweise spärlichen Reizen der inneren Haarzellen wird so eine Fülle an Informationen über die nahezu 100 Milliarden beteiligten Nervenzellen gewonnen. [Els00]

3.3 Frequenzabhängiges Lautheitsempfinden

Eine grundlegende Eigenschaft des Hörsinns ist die unterschiedlich laute Wahrnehmung verschiedener Frequenzenbereiche. Jene, die der Sprachverständlichkeit dienen und damit eine wichtige Rolle spielen, werden empfindlicher wahrgenommen als sehr tief- oder hochfrequente Bereiche. Bereits 1933 wurden hierzu von *Fletcher und Munson* erste Messungen an verschiedenen Probanden durchgeführt und als Kurven gleicher *Lautheit* aufgezeichnet. Der Begriff Lautheit steht dabei für die empfundene Lautstärke. Den Probanden wurden Sinustöne verschiedener Frequenzen über Kopfhörer zugespielt, die sie nach ihrer Empfindung beurteilten. Zwischenzeitlich wurden zahlreiche weitere Messungen durchgeführt, insbesondere auch unter Verwendung von Lautsprechern. Diagramme dieser Art werden oft fälschlicherweise unter dem pauschalen Begriff *Fletcher-Munson-Kurve* geführt, obwohl die Messungen - wegen des Unterschiedes von Kopfhörern zu Lautsprechern - zum Teil gravierende Unterschiede zeigen. Abbildung 3-1 zeigt die Kurven gleicher Lautheit nach dem 2003 überarbeiteten, sowie dem bis dahin gültigen *ISO 226*-Standard. [EP09]

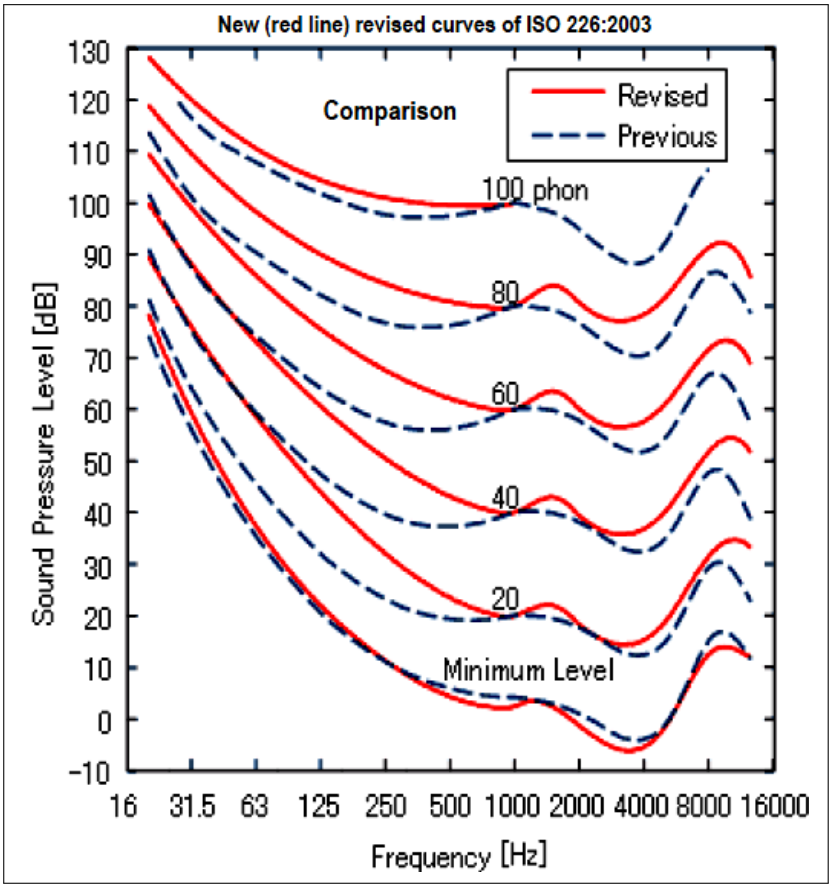

Abbildung 3-1: Kurven gleicher Lautheit nach ISO 226 (Quelle: [Sen05])

Neben den gezeigten Unterschieden bei verschiedenen Abhörlautstärken sind die Kurven zudem bei jedem Menschen unterschiedlich. Des Weiteren spielen Faktoren wie Lebensalter, Tageszeit, Müdigkeit, etc. eine wichtige Rolle. [BA86]

Eine rudimentäre, aber keinesfalls genaue Kompensation dieses frequenzabhängigen Hörempfindens kann durch sog. *Gewichtungskurven* realisiert werden, die auch von einigen Spektralanalysewerkzeugen unterstützt werden. Den größten Bekanntheitsgrad hierbei haben die Bewertungskurven der *DIN IEC 651*-Norm, insbesondere die sog. *A-Bewertungskurve* [Dic87]. Sie dienen in erster Linie der Gewichtung von Messungen und Spezifikationen im Audio-Bereich, erkennbar u.a. durch Zusätze wie „A-weighted“. Beispielsweise wird der Rauschabstand von Audio-Verstärkern oft in gewichteter Form angegeben, was dem tatsächlich empfundenen Rauschabstand näher kommen soll, kritisch betrachtet aber auch als werbewirksame *Beschönigung* der Spezifikationen angesehen werden könnte.

Bei Spektralanalysewerkzeugen, die nicht für Messungen vorgesehen sind, ist der Einsatz solcher Gewichtungen kaum sinnvoll, da sie ohnehin die zahlreichen oben genannten Faktoren nicht ausgleichen können und somit zu weniger Transparenz in der Darstellung führen würden. Umso wichtiger ist es, die Darstellung auch als eine rein technische anzusehen und die Ergebnisse als solche zu bewerten.

3.4 Weitere psychoakustische Aspekte und deren Einfluss

Die im vorangegangenen Abschnitt 3.3 vorgestellten Kurven gleicher Lautheit stellen nur einen von vielen Aspekten aus der Psychoakustik dar, welche zu einer veränderten Wahrnehmung spektraler Anteile führen. Zu den weiteren Aspekten gehören:

- **Bandbreitenabhängigkeit**: Signale geringer Bandbreite, wie z.B. eine Sinuswelle, werden trotz miteinkalkulierter Energiegleichheit leiser wahrgenommen als jene mit größerer Bandbreite. Dieser Zusammenhang ist zudem nicht linear und zeigt sich erst ab einer minimalen, frequenzabhängigen *kritischen* Bandbreite. Dies ist auf die frequenztrennende Funktionsweise des Gehörs zurückzuführen [EP09]. Aus diesem Grund führt eine Filterbank konstanter Güte, wie sie bei der FWT zum Einsatz kommt, auch zu einem dem Gehör besser nachempfundenen Ergebnis als die frequenzlineare Aufteilung der FFT.
- **Simultanverdeckung**: Treffen zwei Töne ähnlicher Frequenz auf das Ohr, so verdeckt der tiefere Ton den höheren, sofern dieser nicht deutlich lauter ist. Letzteres würde dazu führen, dass der tiefere Ton verdeckt wird [BA86]. Die Verdeckung lässt sich im Allgemeinen ebenfalls auf die frequenzselektive Arbeitsweise des Gehörs zurückführen, wobei die Überlappung der Frequenzbereiche - beschrieben im *Leistungsspektrum-Modell von Fletcher* - den Einfluss benachbarter Töne erklärt. Genau betrachtet ist dieses Modell aber nur eine vereinfachte Beschreibung der Vorgänge, welches u.a. die komplexen Abläufe im Gehirn außer Acht

lässt [Moo07]. Obwohl hierzu Algorithmen existieren, die sich diesen Effekt z.B. bei der verlustbehafteten Kompression von Audiosignalen zunutze machen, ist eine Nachbildung dessen im Zuge einer Spektralanalyse in der Praxis nicht vorgesehen.

- **Zeitliche Verdeckung**: Ähnliches wie bei der Simultanverdeckung geschieht, wenn Töne in kurzen Zeitabständen aufeinanderfolgen. Hierbei verdeckt das zuerst eintreffende Schallereignis die unmittelbar darauf folgenden, was als *post-masking* bzw. *Nachverdeckung* bezeichnet wird und auf ein Ausklingverhalten des Gehörs zurückzuführen ist. Außergewöhnlich dagegen ist, dass sich auch vor dem ersten Schallereignis eine Verdeckung bemerkbar macht, welche als *pre-masking* bzw. *Vorverdeckung* bezeichnet wird. Dies kann damit erklärt werden, dass auch das Gehör eine gewisse Zeit zur Verarbeitung von Schallereignissen braucht, innerhalb derer sich Einklingphasen bemerkbar machen [FZ07]. Auch dieser Effekt wird bei der spektralen Darstellung durch Analyzer nicht berücksichtigt. Für einen ausreichenden Informationsgehalt bezüglich der zeitlichen Abläufe wäre das außerdem nur mit einer Wavelet-Transformation umsetzbar, da die FFT eine dafür unzureichende Zeitauflösung aufweist.

- **Grundfrequenzergänzung**: Aufgrund seiner Größe kann das Ohr, wie aus Abbildung 3-1 ablesbar, einzeln auftretende tiefe Frequenzen nur entsprechend stark gedämpft aufnehmen. Klänge, die aus einer Zusammensetzung eines Grundtons und vieler Obertöne bestehen, können dagegen bis zu wesentlich tieferen Frequenzen noch deutlich wahrgenommen werden. Der Grund dafür ist, dass das Gehirn aufgrund von Hörerfahrungen schwache oder sogar fehlende Grundtöne nach dem Muster der Obertöne wiederherstellen kann. Somit kann beispielsweise eine Bassgitarre auch durch ein Küchenradio noch als solche identifiziert und mit verhältnismäßig geringen Qualitätseinbußen wahrgenommen werden. Dieser Effekt wird in der Audiotechnik gezielt ausgenutzt, um einen verstärkten Basseindruck zu erzielen, indem entsprechende Obertöne durch Verzerrung, also z.B. mittels nicht-linearer Verstärkungskennlinien, betont bzw. ergänzt werden. Mit einem Spektrumanalysator ist dieses Phänomen nicht nachbildbar, wobei es entgegengesetzt zur Kurve gleicher Lautheit wirkt und somit einen - wenn auch nicht exakten - ausgleichenden Charakter aufweist. [BA86]

- **Weitere subjektive Wahrnehmungen**: Neben den bereits genannten psychoakustischen Aspekten existieren noch zahlreiche weitere, zum Teil nicht ausreichend genau beschreibbare Aspekte. Dazu gehören u.a. das Verbinden von Klängen mit Erinnerungen und Gefühlen, das sinnesübergreifende Bewerten von Klangereignissen (z.B. wird Schall, der von großen Gegenständen ausgeht, lauter empfunden) und die Erkennung von Mustern durch Hörerfahrung (z.B. wird der Klang eines laut gespielten Tons am Klavier immer als laut identifiziert, auch wenn er auf technischem Wege leise wiedergegeben wird) [EP09]. All diese Wahrnehmungen können nicht durch technische Hilfsmittel sichtbar gemacht werden und zeigen, wie deutlich überlegen der menschliche Hörsinn ist.

4 Spektralanalyse-Plugins im Vergleich

4.1 Einleitung

In diesem Kapitel wird ein kurzer Vergleich verschiedener, derzeit am Markt befindlicher VST-Plugins vorgenommen, welche eine Spektralanalyse unterstützen. Je ein Vertreter aus dem Software- und Freeware-Bereich wird dabei genauer vorgestellt und durch eine Tabelle weiterer Plugins komplettiert.

4.2 Waves PAZ

Das US-amerikanische Software-Unternehmen *Waves Audio Ltd*[2] zählt zu den bekanntesten in diesem Bereich. Obwohl ihr Preis-Leistungsverhältnis zum Teil polarisiert, ist deren Stellenwert als Industriestandard unumstritten. Der *Waves PAZ Psychoacoustic Analyzer* stellt dabei eines der wenigen, wenn nicht sogar das einzige professionelle Spektralanalyse-Plugin dar, das von der Wavelet-Transformation Gebrauch macht [Wav]. Abbildung 4-1 zeigt die Benutzeroberfläche des Plugins. Neben den bei einer Software dieser Art üblichen Parametern, wie z.B. jene zur Anpassung der Darstellung, ist im Funktionsumfang auch der Stereobild-Analysator *Stereo Position Display* und eine sog. *Meter*-Anzeige enthalten.

Die spektrale Auflösung wird über den im Bassbereich gewünschten Detailgrad mit dem „LF res“-Parameter eingestellt [Wav]. Das ist etwas ungewöhnlich, spiegelt aber die interne, wavelet-basierende Verarbeitung wider, bei der aus der Frequenzauflösung im Bassbereich auf die Anzahl der notwendigen Stufen der Filterbank und die damit notwendige Anzahl von Abtastwerten pro Block geschlossen werden kann.

Wenngleich das Plugin eine umfangreiche Ausstattung aufweist, so mangelt es der 2D-Liniendarstellung, die sich als Standard bei Analyzern etabliert hat, etwas an Granularität. Diese fällt trotz der linearen Interpolation zwischen den Datenwerten auf. Zudem wird der Vorteil der guten Zeitauflösung im hochfrequenten Bereich, der durch die Wavelet-Transformation bedingt ist, von der Darstellungsform *verschluckt*, da das Auge den schnellen Änderungen der Linien nur unzureichend folgen kann.

[2] siehe http://www.waves.com

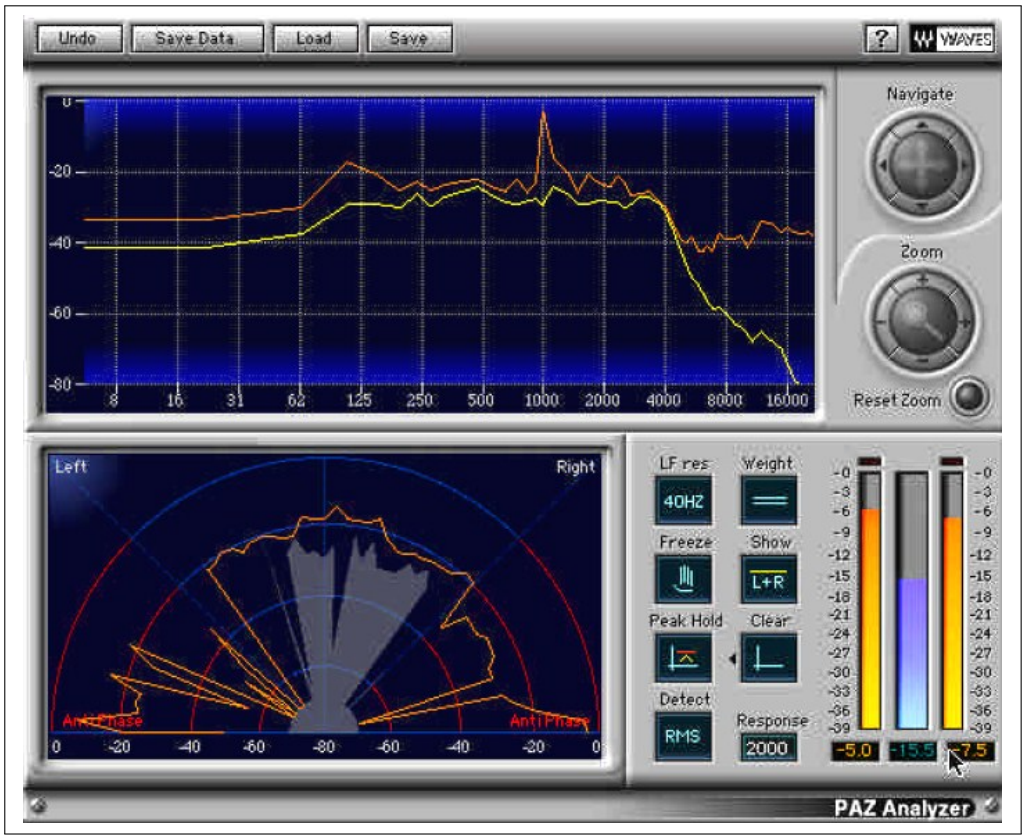

Abbildung 4-1: Waves PAZ (Quelle: [Wav])

4.3 Voxengo SPAN

Das von *Aleksey Vaneev* 2002 gegründete russische Software-Unternehmen *Voxengo*[3] bietet eine umfassende Produktpalette qualitativ hochwertiger Plugins an, die zwar weniger bekannt und verbreitet sind als jene von *Waves*, dafür aber über ein besseres Preis-Leistungsverhältnis verfügen.

Zu den kostenlos erhältlichen Plugins zählt u.a. der Spektrumanalysator *Voxengo SPAN*, dessen Benutzeroberfläche in Abbildung 4-2 dargestellt ist. Obwohl es sich um Freeware handelt, ist der Funktionsumfang mit dem des Waves PAZ vergleichbar.

Neben der üblichen Spektralansicht und den dazugehörigen Einstellmöglichkeiten, bietet das Plugin u.a. auch eine umfangreiche Signalstatistik, *K-Metering*, *Stereo Mid/Side-Analyse* und einen Korrelationsgradmesser zur Kontrolle der Stereobreite. Der wesentliche Unterschied zum *Waves PAZ* ist, dass das Spektrum, wie bei den meisten anderen Plugins, FFT-basierend ermittelt wird. [Van]

[3] siehe `http://www.voxengo.com`

Abbildung 4-2: Voxengo SPAN (Quelle: [Van])

4.4 Vergleichstabelle

Die folgende Tabelle zeigt eine Liste[4] mit Spektralanalyse-Plugins im Vergleich:

Plugin	**Preis (ca.)**	**Kern**	**Stereo-Analyse**	**Besonderheiten**
Blue Cat Audio FreqAnalyst Multi	69 EUR	Filter	nein	Mehrkanalfähig inkl. Verdeckungsanalyse
Darrell Tam DtBlkFx	frei	FFT	nein	Farbdiagramm-Darstellung, Spektraleffektprozessor
iZotope Ozone	250 $	FFT	ja	Mastering-Werkzeug inkl. EQ, Limiter, . . .
Seven Phases Spectrum Analyzer	frei	Filter	nein	Terz-,Halbterzbandanalyse
Vertex DSP MultiInspector	frei(lite), 89 EUR	FFT	nein	Mehrkanal-Terzbandanalyse(frei) Mehrkanal-FFT-Modus
Voxengo SPAN	frei	FFT	ja	Signalstatistiken, *K-Metering*
Waves PAZ	200 $	FWT	ja	wavelet-basierend, *Stereo Position Display*

Tabelle 4-1: Spektralanalyse-Plugins im Vergleich

[4] Recherche-Quelle: http://www.kvraudio.com

5 Aufgabenstellung

5.1 Einleitung

In diesem Kapitel werden die Anforderungen an das zu entwickelnde Plugin analysiert und eine Aufgabenstellung formuliert. Das Plugin soll sich von den in Kapitel 4 bewerteten abheben, gleichzeitig aber auch die gebräuchlichsten Grundfunktionen unterstützen. Dementsprechend werden sowohl Ziele, als auch Nicht-Ziele definiert und entsprechend priorisiert.

5.2 Funktionsumfang

5.2.1 Standardfunktionen

Um einem im Umgang mit Audio-Plugins und Spektralanalyse vertrauten Anwender den Einstieg zu erleichtern, sollten folgende Standardfunktionen unterstützt werden:

- **Signalquelle**: Die Wahl der Signalquelle ist insbesondere dann von Bedeutung, wenn das zu analysierende Audiomaterial in mehrkanaliger Form, also z.B. in Stereo, vorliegt. Üblicherweise kann man zwischen dem linken Kanal, dem rechten Kanal und dem Durchschnitt beider Kanäle, der auch als *Mitte* bezeichnet wird, wählen. Zusätzlich erweist es sich in der Praxis als nützlich, wenn das Seitensignal analysiert werden kann, welches aus der Differenz zwischen dem linken und dem rechten Kanal ermittelt wird.
- **Auflösung**: Die Auflösung spielt bei der Spektralanalyse eine elementare Rolle und zählt daher ebenfalls zu den Standardfunktionen. In den meisten Fällen ist diese direkt in Form der Blockgröße auswählbar, welche einer ganzzahligen Potenz von 2 entspricht. Manche Hersteller ersetzen diesen Parameter durch einen verständlicheren, wie z.B. eine einstellbare *Qualität*. Darauf wird hier aber verzichtet.
- **Fensterung**: Auch dieser Parameter zählt zum Standard FFT-basierender Analysatoren, wenngleich hier immer häufiger versucht wird, dem Benutzer die technischen Details durch automatische Einstellungen zu ersparen. Diese proprietären Lösungen werden aber nicht von allen Benutzern geschätzt, da es diesen oft an Vergleichbarkeit und Transparenz fehlt. Daher ist hier die Entscheidung für die technische Form, d.h. die direkte Wahl der Fensterfunktion, getroffen worden.

- **Logarithmische Darstellungsform**: Neben einer linearen Frequenzdarstellung des Spektrums, die nicht dem Hörempfinden entspricht und daher auch nur technischen Zwecken dient, wird von einem Spektralanalysator auch eine logarithmische Darstellung erwartet. Dies wurde berücksichtigt und sowohl ein Schalter für die Frequenz-, als auch für die Amplitudendarstellung vorgesehen, mit dem zwischen logarithmischer und linearer Darstellung gewechselt werden kann.

5.2.2 Besondere Funktionen und Eigenschaften

Da es kaum möglich ist, mit den ausgereiften, aufwändig designten und z.T. als Freeware erhältlichen Plugins mitzuhalten, soll sich das zu entwickelnde Plugin mit folgenden, möglichst herausstechenden und gleichzeitig nützlichen Funktionen von seinen Mitstreitern abgrenzen:

- **Verschiedene Analyseverfahren**: Die meisten Spektrumanalysatoren basieren auf einem fest implementierten Analyseverfahren, in aller Regel einer FFT. Ausnahmen, wie der in Abschnitt 4.2 vorgestellte *Waves PAZ*, basieren intern auf einer FWT. Terzbandanalyzer, welche durch ihr Balkendiagramm und ihre konstante Antwort auf *Rosa Rauschen* bekannt sind, arbeiten intern häufig mit Filterbänken konstanter Güte, was aber wiederum einer FWT nahe kommt. Trotz umfangreicher Recherche konnte kein Plugin gefunden werden, das sowohl FFT und FWT, als auch darüber hinaus Wavelet-Pakete mit wählbarer oder adaptiver Basis unterstützt. Genau in diesem Punkt kann sich das zu entwickelnde Plugin deutlich von anderen abheben, weshalb hierin auch der Schwerpunkt gelegt wurde.

- **Farbdiagramm-Darstellung**: Die Farbdiagramm-Darstellung, welche auf der X-Achse die Zeit, auf der Y-Achse die Frequenz und als Farbton den Betrag der Amplitude darstellt, ist bei Spektralanalysatoren seltener anzutreffen als die klassische Liniendarstellung, welche aber den zeitlichen Verlauf des Spektrums nicht wiedergibt. Nachteilig ist, dass der Betrag der Amplitude nur sehr ungenau aus dem Farbton ablesbar ist. Des Weiteren ist aufgrund der schlechten Zeitauflösung FFT-basierender Analyzer eine getrennte Darstellung der zeitlichen Entwicklung wenig informativ und daher verzichtbar.
  Diese Darstellung eignet sich aber gut, um die Unterschiede zwischen den verschiedenen Analyseverfahren zu zeigen und verbessert darüber hinaus auch die Lesbarkeit von Spektralanalyseergebnissen höherer Zeitauflösung, welche z.B. durch eine FWT ermittelt wurden.
  Zudem wird die Unschärfe der Farbtondarstellung zum Anlass genommen, um eine Normierung der Amplituden durchzuführen. Das Spektrum wird somit, unabhängig von der Lautstärke des Signals, bei gleicher relativer Zusammensetzung auch immer gleich dargestellt. Der Vorteil ist, dass sowohl bei ruhigen, als auch bei lauten Musik-Passagen das Spektrum immer unter voller Ausnutzung der Am-

plitudenauflösung dargestellt wird und somit ein *Zoomen* der Amplitudenachse nicht notwendig ist.

- **Testsignalgenerator**: Um das Plugin effizienter testen zu können, wurde ein einfacher Testsignalgenerator integriert, der eine Analyse auch ohne anliegendes Signal ermöglicht. Diese Zusatzfunktion ist nur für die Entwicklung und Demonstration der Funktionsweise von Bedeutung und damit kein wesentlicher Teil des Plugins.

5.2.3 Echtzeit

Bislang wurde noch nicht geklärt, was unter dem Begriff *Echtzeit* zu verstehen ist, und in welchem Kontext dieser zur Spektralanalyse steht.

Unter *Echtzeit* im engeren Sinne versteht man die Reaktion auf Ereignisse innerhalb einer festgelegten Zeitspanne. Dabei stellt die Schnelligkeit nur ein Mittel dar, um Rechtzeitigkeit zu erzielen. Voraussetzung dafür ist ein determiniertes System, da ansonsten Zeitbedingungen weder geprüft noch garantiert werden können. Je nachdem, welche Konsequenzen eine Überschreitung der Zeitfrist nach sich zieht, spricht man von *harter* oder *weicher Echtzeit* [Zöb08].

In Zusammenhang mit dem hier entwickelten Plugin ist der Begriff *Echtzeit* in einem weitläufigeren Kontext zu verstehen. Alleine dadurch, dass die Entwicklung nicht für ein fest vorgegebenens System erfolgt, sondern auf verschiedenste, universelle Soft- und Hardware-Komponenten aufbaut, kann bezüglich der Erfüllung von Zeitbedingungen keine Aussage getroffen werden. Unter *Echtzeitspektralanalyse* wird hier also lediglich die laufende Analyse eines *Audiostreams* verstanden, womit das Spektrum und dessen Veränderungen während des Abspielens der Audiosignale dargestellt werden können.

5.2.4 Nicht-Ziele

Nachdem in den letzten beiden Abschnitten 5.2.1 und 5.2.2 die Ziele in Form der zu implementierenden Funktionen festgelegt wurden, sollen nun auch die Nicht-Ziele definiert werden. Diese sind wichtig, um eine klare Abgrenzung zwischen jenen Projektbestandteilen zu schaffen, die je nach Sichtweise als Projektinhalt oder -umfeld interpretierbar sind und somit für Unklarheiten sorgen könnten.

Folgende Nicht-Ziele wurden festgelegt:

- **Marktreife**: Das Plugin soll in erster Linie die grundlegenden Möglichkeiten verschiedener Analyseverfahren zeigen, Marktreife steht dabei nicht im Fokus. Ein aufwändiges Oberflächen-Design, eine optimierte Bedienbarkeit, eine umfangreiche Hilfe und umfangreiche Zusatz- und Schnittstellenfunktionen sind aufgrund begrenzter Entwicklungszeit nicht umsetzbar.
- **Kompatibilität und Plattformunabhängigkeit**: Von einem VST-Plugin ist zwar ein einhergehendes Mindestmaß an Kompatibilität zu erwarten, allerdings wären hier aufwändige Praxistests nötig, um sicherzustellen, dass es sich tatsächlich fehlerfrei und stabil in gängige Musikprogramme sämtlicher Plattformen einbinden lässt.
- **Stabilität und Performance**: Langzeit- und Lasttests, sowie genaue Tests der internen Abläufe, die als *White-Box-Tests* bezeichnet werden, wären für eine Garantie bezüglich Stabilität und Performance wichtig. Da das Plugin aber vorerst nicht für die Vermarktung gedacht ist und solche Tests sehr aufwändig sind, werden diese nur eingeschränkt durchgeführt. Damit verbundene Probleme müssen in der ersten Version in Kauf genommen werden.
- **Messgenauigkeit**: Obwohl der aktuelle Trend bei Spektralanalyse-Plugins immer mehr in Richtung integrierter Messfunktionen zu gehen scheint, ist dieses Plugin eher für musikalische und experimentelle Zwecke gedacht, was sich auch in der Darstellungsform widerspiegelt.

5.2.5 Priorisierung

Während der Projektphase und speziell während der Umsetzung ist es von Vorteil, wenn neben der Festlegung von Zielen und Nicht-Zielen auch eine Priorisierung, also eine Reihung nach Wichtigkeit, vorgenommen wird. Dadurch lassen sich die Risiken eingrenzen und frühzeitig grobe Abweichungen von Kosten, Zeit und Leistung erkennen. Unter engem Kosten- und Zeitdruck müssen somit bei Verzug nur Abstriche in den weniger wichtigen Teilen des Projektumfangs in Kauf genommen werden, was wiederum gegenüber dem Auftraggeber leichter argumentierbar ist.

Die Priorisierung wurde, beginnend mit den wichtigsten Punkten, wie folgt festgelegt:

1. Analyseverfahren FFT und FWT
2. Darstellung des Spektrums als Farbdiagramm
3. Lineare oder logarithmische Frequenz- und Amplitudendarstellung
4. Wahl der Signalquelle und Auflösung

5. Wählbare Fensterung für FFT
6. Auswahl verschiedener Wavelets (für FWT)
7. Analyseverfahren WPT
8. Auswahl der Knotenebene (für WPT)
9. Adaptive Analyse mittels *Best-Base*-Algorithmus (für WPT)
10. Testsignalgenerator

5.2.6 Zusammenfassung

Das zu entwickelnde VST-Spektralanalyse-Plugin soll parallel zur FFT auch andere, wavelet-basierende Analyseverfahren unterstützen, um es von anderen dieser Art abzugrenzen.

Neben der klassischen Wavelet-Transformation soll auch die WPT unterstützt werden. Letztere lässt sich über die Wahl der Knoten variieren. Der Benutzer kann dabei zwischen Knotenebenen wählen, oder die Knotenwahl einem Algorithmus überlassen, der automatisch die dem Signal entsprechende, beste Basis adaptiv ermittelt.

Die Darstellung des Spektrums erfolgt in Form eines Farbdiagramms. Der Anwender soll dabei zwischen linearer und logarithmischer Darstellung sowohl der Frequenz als auch der Amplitude wählen können. Der Schwerpunkt liegt in der Implementierung der Spektralanalyse und unmittelbar damit zusammenhängenden Funktionen und Parametern. Aspekte wie Oberflächendesign, Kompatibilität, etc., die für die Marktreife notwendig wären, werden aus dem Projektumfang ausgegrenzt.

6 Konzept

6.1 Einleitung

In diesem Kapitel wird das Konzept des Spektralanalyse-Plugins erarbeitet. Zu Beginn wird die VST-Schnittstelle vorgestellt und mit anderen Schnittstellen dieser Art verglichen. Anschließend werden Frameworks diverser Plattformen bewertet, wodurch die Entscheidung für das am besten geeignete Framework begründet werden kann. In gleicher Weise werden auch unterschiedliche Bibliotheken vorgestellt. Nach Festlegung der Rahmenbedingungen wird näher auf das Software-Design eingegangen. Dabei werden sowohl allgemeine Prinzipien, als auch deren Anwendung erläutert. Die Vorgehensweise wird anhand eines Beispiels detailliert beschrieben. Dem gleichen Prinzip folgende Design-Aspekte werden daraufhin nur mehr grob skizziert. Das Kapitel wird mit der genauen Untersuchung eines Software-Architekturaspektes abgeschlossen, welcher für die Leistung der Software von besonderer Bedeutung ist.

6.2 VST

Die virtuelle Studio Technologie - kurz VST™- wurde vom deutschen Unternehmen *Steinberg* 1996 zusammen mit dem Produkt *Cubase VST* veröffentlicht. Die Grundidee dabei war und ist, dass nach dem Vorbild eines realen Tonstudios auch im Rechner einzelne, virtuelle Signalverarbeitungsmodule wie EQ's, Halleffekte, etc. flexibel an beliebigen Stellen im virtuellen Mischpult platziert werden können. Die Veröffentlichung des Schnittstellenaufbaus in Form eines sog. *Software Development Kit (SDK)* ermöglichte es schließlich auch Drittanbietern, Plugins für diese Umgebung zu entwickeln [Ste]. Aus technischer Sicht handelt es sich um sog. *Dynamic Link Libraries (DLL)*, die zur Laufzeit in (Windows-)Programme eingebunden werden können.

Mittlerweile hat sich VST als Standard etabliert, was sich durch die Anzahl[5] der dafür verfügbaren Plugins und der unterstützenden Audioprogramme begründen lässt. Daher ist diese Schnittstelle für die Entwicklung des Spektralanalyse-Plugins gewählt worden.

Zu den Mitstreitern von VST zählen u.a. *Real-Time AudioSuite (RTAS)* vom *Pro Tools*-Hersteller *Avid Technology* (früher *Digidesign*), *Audio Units (AU)* von *Apple* und *DirectX*

[5] Ausgewertet über http://www.kvraudio.com am 10. April 2011

von *Microsoft*, die ebenfalls von zahlreichen Plugins unterstützt werden.
Hürden zwischen Plugin-Formaten können mit sog. *Plugin-Wrappern* überwunden werden, welche auch als Software-Adapter angesehen werden können. Diese ermöglichen es, dass z.B. ein VST-Plugin als DirectX-Plugin eingesetzt werden kann. Plugin-Wrapper sind allerdings nicht für alle Schnittstellenformate erhältlich und stoßen, speziell die plattform-übergreifende Verwendung betreffend, rasch an ihre technischen Grenzen.

6.3 Frameworks

Im vorangegangenen Abschnitt wurde die Entscheidung für die VST-Schnittstelle begründet. Nun stellt sich die Frage, welches Framework für die Entwicklung am besten geeignet ist.

Bei einem *Framework* handelt es sich um Software, die immer wiederkehrende Elemente kapselt und einen grundlegenden Design- und Architekturrahmen vorgibt [Pas01]. Frameworks werden daher nicht für eine bestimmte Anwendung geschrieben und sind auch nicht als lauffähige Programme für Endanwender anzusehen. Sie unterstützen vielmehr die Entwicklung in dem dafür vorgesehenen Anwendungsbereich und sorgen durch gezielte Mechanismen dafür, dass grundlegende Design- und Architekturvorgaben eingehalten und weitergetragen werden. Des Weiteren zeichnen sich Frameworks durch eine universelle und unabhängige Einsetzbarkeit aus, welche sie unter Verwendung von Mitteln, wie z.B. Abstraktionen, erzielen. In Tabelle 6-1 wird eine Übersicht einiger *Open Source*-Frameworks und SDK's gezeigt, welche die Entwicklung VST-fähiger Audio-Plugins unterstützen:

Framework/SDK	Entwickler	Plattformen	Sprache	Besonderheiten
WDL [6]	Cockos Incorporated	Windows, Mac OS X	C++	nativ
Delphi VST SDK [7]	Frederic Vanmol	Windows, Mac OS X	Delphi	Übersetzte Basisdefinition
jVSTwRapper [8]	Daniel Reinert	Windows, Mac OS X, Linux	Java	Laufzeitumgebung
JUCE [9]	Raw Material Software	Windows, Android Mac OS X, Linux	C++	nativ
JUCED [10]	Community	Windows, Android Mac OS X, Linux	C++	Basiert auf JUCE, nativ
VST.NET [11]	Marc Jacobi	Windows	C#	Laufzeitumgebung
VST-SDK [12]	Steinberg	Windows, Mac OS X	C	Basisdefinition, prozedural

Tabelle 6-1: VST-Frameworks verschiedener Plattformen

6.3.1 VST-SDK

Die Basis für die Entwicklung eines VST-Plugins ist die Schnittstellenspezifikation, welche in Form des VST-SDK von Steinberg zur Verfügung gestellt wird. Das SDK, insbesondere die bereits abgelöste, aber weiterhin als Standard geltende Version 2.x, beinhaltet lediglich eine in C++ geschriebene, aber großteils prozedural umgesetzte Schnittstellenvorgabe.

Wenngleich das VST-SDK die Basis für die Entwicklung eines jeden VST-Plugins bilden muss, so eignet es sich nicht als direkter Ausgangspunkt. Das gilt auch für in andere Sprachen übersetzte Varianten. Aufgrund der rein funktionell orientierten, prozeduralen Umsetzung würde eine unmittelbare Verwendung schnell zu einem unübersichtlichen und kaum wartbaren Code führen.

6.3.2 Laufzeitumgebungen

Einen guten Ausgangspunkt für die Entwicklung von VST-Plugins stellen speziell dafür vorgesehene Frameworks dar. Diese bauen zwar zur Einhaltung der Schnittstellen-Definitionen auf das VST-SDK auf, verbergen es aber nach außen hin, womit alle irrelevanten Details ausgeblendet werden. Darüberhinaus stellen VST-Frameworks hilfreiche Klassen und Bibliotheken zur Verfügung, die ein objektorientiertes Design verfolgen und somit eine gute Grundlage für die Entwicklung bieten.

Neben diesen Merkmalen können mithilfe von Frameworks auch Brücken zu anderen Plattformen realisiert werden. Ein Beispiel dafür sind jene Frameworks, welche eine Entwicklung auf sog. *Laufzeitumgebungs*-Plattformen ermöglichen. Zu den bekanntesten Plattformen zählen *Java™* von *Sun Microsystems* und *.NET™* von *Microsoft*. Sie zeichnen sich dadurch aus, dass der Quellcode nicht direkt in Maschinenbefehle, sondern in einen universellen Zwischencode - auch *Byte-Code* oder *managed code* genannt - übersetzt wird. Über ein Programm, das als virtuelle Maschine bezeichnet wird und für unterschiedliche Plattformen erhältlich ist, kann dieser Zwischencode dann ausgeführt werden [Fla03]. Das bedeutet, dass es im Idealfall möglich ist, ein Programm zu schreiben, das auf den verschiedensten Betriebssystemen ausgeführt werden kann, ohne dass Änderungen daran vorgenommen werden müssen. In der Praxis ist das

[6] siehe http://cockos.com/wdl
[7] siehe http://www.axiworld.be/vst.html
[8] siehe http://jvstwrapper.sourceforge.net
[9] siehe http://www.rawmaterialsoftware.com/juce.php
[10] siehe http://code.google.com/p/juced
[11] siehe http://vstnet.codeplex.com
[12] siehe http://www.steinberg.net/en/company/developer.html

allerdings, bedingt durch verschiedene Versionen und Grenzen der virtuellen Maschinen, mit Einschränkungen verbunden. Detaillierte Kenntnisse über die Handhabung der Betriebssystem-API's sind jedenfalls nicht mehr notwendig, womit die Entwicklung erheblich erleichtert und beschleunigt wird.

Der Nachteil der Laufzeitumgebungsarchitektur ist, dass der Zwischencode während der Ausführung interpretiert und in die jeweilige Maschinensprache übersetzt werden muss. Obwohl der Zwischencode dem Maschinencode nachempfunden ist und moderne virtuelle Maschinen mit fortschrittlichen Optimierungen arbeiten, kann die Leistung eines nativ kompilierten Programms nicht erreicht werden. Für rechenintensive Anwendungen ist diese Architektur daher ungeeignet. Aus diesem Grund wurde auf den Einsatz von Laufzeitumgebungsarchitekturen verzichtet.

6.3.3 Native Frameworks

Nachdem in den vorangegangenen Abschnitten 6.3.1 und 6.3.2 die rein auf das SDK basierende Entwicklung, sowie die Nutzung von Laufzeitumgebungen beleuchtet wurden, sind schließlich noch native Frameworks zu untersuchen, mit welchen sich hohe Geschwindigkeiten erzielen lassen. Dazu zählen im Wesentlichen *WDL* von *Cockos Incorporated* und *JUCE* von *Raw Material Software*. Zu letzterem existiert zudem noch die Erweiterung *JUCED*.

Bei dem von Julian Storer entwickelten Framework *JUCE*, das unter der *GNU General Public Licence - Version 2* für die Entwicklung ebenfalls offengelegter Software verwendet werden kann, handelt es sich um ein ausgereiftes und äußerst umfassendes Framework, das die Entwicklung auf zahlreichen Plattformen, mitunter Windows, Mac OS und Linux, unterstützt. Für eine reibungslose Integration sorgen zahlreiche Vorlagen, welche das Einbinden des Frameworks in unterschiedlichste Entwicklungsumgebungen erleichtern. Zudem lässt sich das Framework nicht nur durch statisches oder dynamisches Linken einbinden, sondern auch durch das Hinzufügen weniger, direkt in das Projekt integrierbarer *Header-Dateien*. Zum Umfang gehören mitunter zahlreiche Klassen zur Gestaltung der Benutzeroberfläche, welche durchaus z.B. mit *AWT* aus dem Java-Bereich vergleichbar sind. Zur Unterstützung kann der mitgelieferte Oberflächen-Designer *Jucer* eingesetzt werden. Neben der umfangreichen Benutzeroberflächenbibliothek bietet das Framework auch eine umfassende Unterstützung in anderen wichtigen Bereichen, wie u.a. der Audioverarbeitung.

Das Framework *JUCED* baut auf JUCE auf und verfügt über zahlreiche, nützliche Erweiterungen zu den bereits schon umfangreichen *JUCE*-Klassen. Dazu zählen weitere GUI-Elemente wie z.B. ein Joystick, eine Abstraktion der VST-Parameter und vieles

mehr. Da an der Weiterentwicklung gemeinschaftlich gearbeitet wird, werden laufend Verbesserungen und Erweiterungen vorgenommen. Der Nachteil dieser Philosophie dagegen ist, dass die Qualität der Erweiterungen sehr unterschiedlich ist. Außerdem ist eine fehlerfreie Kompilierung auf allen Plattformen nicht immer sichergestellt, wodurch mit einem zusätzlichen Test- und Integrationsaufwand gerechnet werden muss. Dieser ist nur dann zielführend, wenn die Erweiterungen unbedingt benötigt werden bzw. deren Eigenentwicklung wesentlich mehr Aufwand bedeuten würde.

Das unter der proprietären *WDL licence* stehende Framework *WDL* von *Cockos Incorporated* unterscheidet sich in einigen Punkten deutlich von *JUCE*. Als erstes fällt auf, dass es nicht als Gesamteinheit auftritt, sondern vielmehr aus einer Sammlung verschiedener Bibliotheken zusammengesetzt ist. Eine davon ist *IPlug*, welche das eigentliche VST-Plugin-Framework darstellt. Das Zeichnen der Oberfläche wird über die Graphikbibliothek *lice* abgewickelt. In dieser Form existieren noch einige Bibliotheken, die zum Teil über sehr tiefgreifende Funktionen verfügen. Das gesamte Framework ist somit, im Vergleich zu *JUCE*, nicht ganz so einheitlich und ausgewogen, verfügt aber in speziellen Bereichen über mehr Funktionalität, wie z.B. einen integrierten *Pitch-Shifter*, welcher zur Anpassung der Tonhöhe von Audiosignalen eingesetzt werden kann. Die Integration wird zwar auf Basis von Beispielprojekten erleichtert, diese stehen aber vorwiegend nur für *Microsoft Visual Studio*^{TM} zur Verfügung.

Die Weiterentwicklung verläuft derzeit etwas unkontrolliert, da neben dem still liegenden Original bereits mehrere inoffizielle Modifikationen existieren. Initiativen, die das Projekt in geordnete Bahnen lenken sollen, sind in Planung, befinden sich aber noch in der Anfangsphase.

6.3.4 Framework-Entscheidung

In Abschnitt 6.3.1 wurde bereits begründet, warum eine rein auf das SDK basierende Entwicklung nicht empfehlenswert ist. Ohne weitere Abstraktion würde die Entwicklung, auch bei kleineren Projekten, schnell einen Punkt erreichen, an dem der Code nicht mehr wartbar wäre.

Die in Abschnitt 6.3.2 vorgestellten Frameworks zur Entwicklung von VST-Plugins auf laufzeitbasierenden Plattformen würden, aufgrund ihrer guten objektorientierten Umsetzung, dagegen Abhilfe schaffen, können aber nicht mit der Leistung nativ kompilierter Anwendungen mithalten.

Schließlich wurden in Abschnitt 6.3.3 jene verbleibenden, nativen Frameworks vorgestellt, die sich sowohl durch hohe Geschwindigkeit, als auch durch eine gute Entwick-

lungsunterstützung auszeichnen. Besonders auffallend war dabei das *JUCE*-Framework, das auch im kommerziellen Bereich - dort aber kostenpflichtig - eingesetzt wird und daher in puncto Qualität und Leistung heraussticht.

Aus diesem Grund wurde *JUCE* für die Basis der Entwicklung gewählt.

6.4 Bibliotheken

Neben der Wahl eines Frameworks, welche aufgrund von Lizenzkosten, Schulungskosten, etc. oft nur eingeschränkt möglich ist, ist es in der Evaluationsphase eines Softwareprojektes von Vorteil, zu untersuchen, ob Teile der zu entwickelnden Software bereits in Form von Bibliotheken existieren. Insbesondere in der objektorientierten Programmierung, in der viel Wert auf Wiederverwendbarkeit gelegt wird, lässt sich so oft Zeit-, Entwicklungs- und Testaufwand einsparen. Dem gegenüber stehen Lizenzkosten, die es abzuwägen gilt. Letztere spielen derzeit aber keine Rolle, da eine kommerzielle Nutzung vorerst nicht geplant ist.

Obwohl die objektorientierte Programmierung für die Integration externer Bibliotheken prädestiniert ist, müssen diese in das bestehende Design eingegliedert werden. Dazu existieren zwar Entwurfsmuster, die diesen Schritt erleichtern, trotzdem sollte für diese Aufgabe genügend Zeit eingeplant werden. Denn auch die Nutzung einer Bibliothek bedarf Einarbeitungszeit, welche in die Entscheidung miteinzubeziehen ist.

6.4.1 FFT-Bibliotheken

Auf dem Gebiet der schnellen Fouriertransformation existieren unzählige Bibliotheken, wodurch eine genaue Untersuchung jeder Einzelnen unverhältnismäßigen Aufwand bedeuten würde. Aus diesem Grund wird hier nur eine Bibliothek näher beleuchtet.

Im Bereich der freien, unter der *GNU General Public License* stehenden Bibliotheken sticht *FFTW*[13] in puncto Bekanntheit, Reputation und Performance besonders heraus. Diese von M.Frigo und S.Johnson am *Massachusetts Institute of Technology (MIT)* entwickelte, native Bibliothek macht sich nicht nur zur Verfügung stehende Hardwareunterstützungen, wie den erweiterten Prozessorbefehlssatz SSE [FJ], zu Nutze, sondern ermittelt auch zur Laufzeit einen *Plan* für die beste Zusammensetzung sog. *Codelets*, um die Ausführungszeit zu minimieren [FJ98].

[13] siehe http://www.fftw.org

Wenngleich der empirische Vergleich der Leistung unterschiedlicher FFT-Bibliotheken gegenüber *FFTW* [FJ] etwas in die Jahre gekommen ist, so ist die Leistung für den hier zu entwickelnden Analyzer mehr als ausreichend. Auch die Integration erweist sich als einfach, insbesondere aufgrund der guten Dokumentation. Da die Bibliothek als Quellcode zur Verfügung steht, kann sie direkt oder vorkompiliert eingebunden werden. Letzteres kann statisch oder über eine zusätzliche DLL dynamisch erfolgen. Aufgrund der vielen Bestandteile empfiehlt sich während der Entwicklungsphase, die Bibliothek als vorkompilierte DLL einzubinden, um Compile- und Linkzeiten zu minimieren. Statisches Linken ist für das Erstellen der finalen Version empfehlenswert, um die Auslieferung und Installation zu vereinfachen, da auf diesem Wege die zusätzliche DLL wegfällt.

6.4.2 Wavelet-Bibliotheken

Zur Realisierung der Wavelet-Transformation existieren mittlerweile ebenfalls zahlreiche Bibliotheken. Der Schwerpunkt liegt hier aber in produktspezifischen Erweiterungen, wie z.B. für Matlab™. Aber auch für C, C++ oder Java lassen sich Bibliotheken finden, wenngleich deren Implementierungen sich auf den funktionellen Aspekt konzentrieren und somit in puncto Leistung und Optimierung nicht mit jenen der FFT vergleichbar sind.

	Wave++[14]	**bearcave**[15]	**WAT**[16]
Entwickler	S.E.Ferrando, L.A.Kolasa, N.Kovacevic	I.Kaplan	A.Sazonov, A.Stadnik, S.Klimenko
Stand	2000	2002	2001
Dokumentation	PDF nicht durchgängig	HTML (doxygen), ausführlich	TXT, unzureichend
Beispiele	ja	nein	nein
Code	prozedural	objektorientiert	objektorientiert
Wavelets	Beylkin, Coifman, Daubechies, Vaidyanathan	Haar, Daubechies(4)	Daubechies, Gauss, Meyer, Biorthogonal
FWT/IFWT	ja	ja	ja
WPT/IWPT	ja	ja	nein
Beste Basis	ja	ja	nein
Lifting	nein	ja	ja
Besonderheiten	(a)periodisch, effizient durch QMF	universell, nur 2 Wavelets	Signalgenerierung, Zeichnen v. Funktionen

Tabelle 6-2: 1D-Wavelet-Bibliotheken für C/C++ (Vgl [DGQ08])

[14] siehe http://www.scs.ryerson.ca/~lkolasa/CppWavelets.html
[15] siehe http://www.bearcave.com/misl/misl_tech/wavelets/index.html
[16] siehe http://www.phys.ufl.edu/ligo/wavelet

In Tabelle 6-2 sind drei Wavelet-Bibliotheken im Vergleich dargestellt. Die deutliche Einschränkung der Anzahl ist auf folgende Gründe zurückzuführen:

- **Freie Lizenz**: Da keine kommerzielle Nutzung geplant ist, kommen ausschließlich freie Bibliotheken in Frage.
- **C, C++**: Um den Quellcode nicht unter hohem Aufwand in eine andere Sprache portieren zu müssen, wurden C bzw. C++ vorausgesetzt.
- **1D-Unterstützung**: Viele Bibliotheken sind für eine rein zweidimensionale Verarbeitung z.B. im Bereich der Bildbearbeitung vorgesehen und bieten von Haus aus keine Funktionen für die Verarbeitung eindimensionaler Audiosignale. Diese Bibliotheken könnten zwar für die eindimensionale Berechnung zweckentfremdet werden, die negativen Auswirkungen auf die Lesbarkeit und Qualität des Codes würden diesen Schritt aber nicht rechtfertigen.

Eine weitaus umfangreichere Vergleichstabelle kann aus [DGQ08] entnommen werden. Inhaltlich ist diese nicht mehr auf dem neusten Stand, leistet aber als Ausgangspunkt für Recherchen gute Dienste.

Aufgrund der großen Anzahl unterstützter Wavelets, der effizienten Umsetzung und der aufschlussreichen Beispiele wurde die Bibliothek *wave++* von *S.E.Ferrando*, *L.A.Kolasa* und *N.Kovacevic* für die Wavelet-Transformation gewählt. Die zwar nicht durchgängige, aber in den wesentlichen Punkten ausreichende Dokumentation erwies sich, in Kombination mit den Beispielen, als hilfreich. Die Integration ist aufgrund der fehlenden *Namespace*-Deklaration, die in C++ zur Unterscheidung gleich benannter Bezeichner dient, in unveränderter Form als vorkompilierte Bibliothek zwar nicht möglich, das direkte Einbinden des Quellcodes funktioniert aber nach geringfügigen Anpassungen problemlos. Die Bibliothek unterstützt die FWT, die WPT und enthält dazu noch einen Algorithmus zur Ermittlung der besten Basis, welcher in einem Rauschunterdrückungs-Beispiel dokumentiert ist.

Ein Nachteil von *wave++* ist der fehlende Lifting-Algorithmus. Die daraus resultierenden Einschränkungen sind aber gering, da die Transformation mittels QMF bereits sehr effizient implementiert ist und weitere Spezialitäten von Lifting, wie z.B. ein linearer Phasengang mittels biorthogonaler Wavelets, funktionell und qualitativ kaum zu erwartende Verbesserungen der Spektraldarstellung zur Folge hätten. Als größerer Nachteil könnte die prozedurale, zum Teil schwer lesbare Implementierung der Bibliothek genannt werden. Aber auch das ist nebensächlich, da ohnehin eine Abstraktion der Transformation im Design vorgesehen werden muss, um u.a. auch die FFT, die mittels *FFTW* berechnet wird, zu integrieren. Durch diese Vorgehensweise ist es zudem möglich, für eine spätere Erweiterung weitere Bibliotheken einzubinden und auf diesem Wege z.B. die nachträgliche Integration von Lifting vorzusehen.

Die *bearcave*-Bibliothek von *Ian Kaplan* zeichnet sich zwar durch ihre sehr gelungene Dokumentation in Form eines Tutorials aus, wurde aber aufgrund der wenigen unterstützten Wavelets nicht gewählt. Es handelt sich dabei eher um eine Sammlung einzelner Klassen, die nicht als einheitliche Bibliothek auftreten und vielmehr die praktische Umsetzung veranschaulichen sollen. Auch wenn diese Klassensammlung schließlich nicht gewählt wurde, so erwies sich das Tutorial dennoch als sehr aufschlussreich.

WAT von *Sergey Klimenko*, *Andrei Sazonov* und *Alexei Stadnik* war aufgrund der mangelhaften Dokumentation nicht einschätzbar und wurde aus diesem Grund auch nicht gewählt. Die notwendige Einarbeitungszeit bis zur Erstellung eines ersten Testbeispiels würde einen Mehraufwand bedeuten, der nur bei fehlenden Alternativen eine Option wäre.

6.5 Prinzipieller Aufbau

In den vorangegangenen Abschnitten wurde die Wahl der externen Komponenten getroffen. Nun kann der prinzipielle Aufbau, wie in Abbildung 6-1 dargestellt, festgelegt werden.

Den Einstiegspunkt stellt die VST-Schnittstelle dar, die durch das Framework einfach und intuitiv implementiert werden kann. Sie stellt die Verbindung zum sog. *Host* her, welcher die aufrufende Audiosoftware repräsentiert, und erfüllt dabei zwei wichtige Aufgaben: Zum einen werden Informationen bezüglich der Parameter und Einstellungen des Plugins ausgetauscht, die von beiden Seiten gelesen und modifiziert werden können. Zum anderen realisiert das Plugin die Audiosignalverarbeitung, indem eine speziell dafür vorgesehende Methode implementiert wird. Die Audiodaten werden dabei blockweise verarbeitet. Da die Blockgröße von außen - also dem Host - bestimmt wird, muss die Implementierung davon unabhängig sein, was mittels eines Buffers erzielt wird.

Die Transformation muss im Zuge des Designs abstrahiert werden, damit sich verschiedene Algorithmen und Bibliotheken unabhängig voneinander integrieren lassen und deren Details nach außen hin, z.B. gegenüber der aufrufenden Audiosignalverarbeitung, verborgen werden. Ähnliches gilt auch für die dafür zum Einsatz kommenden Wavelets, sowie für die Fensterung.

Bevor das Spektrum angezeigt werden kann, müssen dessen Daten, welche in unterschiedlichen Formaten, Reihenfolgen und Aufteilungen vorliegen, noch aufbereitet werden. Da auch hier nicht sichergestellt werden kann, dass die Daten sofort weiterverarbeitet werden können, ist an dieser Stelle ebenfalls ein Buffer vorgesehen.

Den letzten, aber nicht unwesentlichen Teil der Kette stellt die Benutzeroberfläche dar, über welche das Spektrum angezeigt wird und der Benutzer verschiedene Einstellungen, wie z.B. die Wahl der Transformationsart, vornehmen kann. Auch hier hat das Framework einen wesentlichen Einfluss auf die Gestaltungsmöglichkeiten, da es sämtliche Bedienelemente zur Verfügung stellt und das Zeichnen auf der Oberfläche abwickelt.

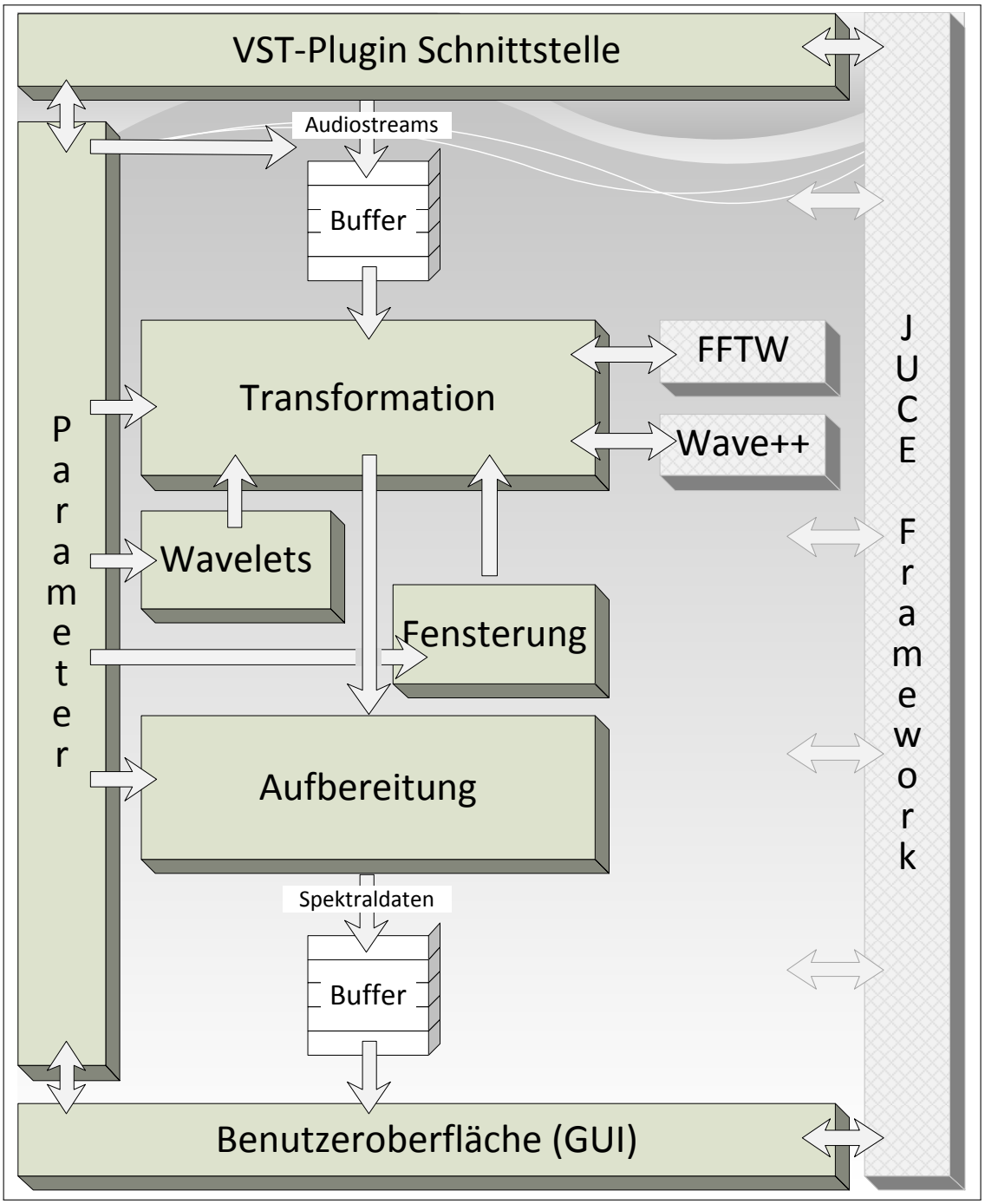

Abbildung 6-1: Prinzipieller Aufbau des Plugins

6.6 Softwaredesign

6.6.1 Einleitung

Das Softwaredesign ist einer der wichtigsten Punkte in der Softwareentwicklung. Es hat einen erheblichen Einfluss auf die spätere Umsetzung und trägt entscheidend dazu bei, dass der Code auch nach einigen Änderungen noch übersichtlich, erweiterbar, testbar und leistungsfähig bleibt. Fehler oder Einsparungen an dieser Stelle führen häufig zu unvorhersehbaren Terminverschiebungen und zu unverhältnismäßig stark steigendem, dem Auftraggeber schwer erklärbarem Aufwand bei Wartungen und Erweiterungen.

6.6.2 Klassische versus agile Softwareentwicklung

In der Praxis hat sich gezeigt, dass im ersten Design nicht alle Anforderungen berücksichtigt werden können, da sich manche erst im Laufe der Umsetzung oder im Zuge einer späteren Erweiterung ergeben. [Kel08]

Um eine größere Flexibilität bezüglich nachträglicher Design-Anpassungen zu erzielen, ist das zyklische Schreiben und Ausführen von Tests während der Umsetzung empfehlenswert [SW07]. Dies ist wiederum nur dann mit vertretbarem Aufwand möglich, wenn bereits im Design der Grundstein für die Testbarkeit gelegt wird. Darauf wird in Abschnitt 6.6.4 noch genauer eingegangen. Durch das zyklische Testen kann zeitnah erkannt werden, ob gegenwärtige Code-Änderungen funktionieren, und ob diese sich negativ auf bereits bestehende Programmteile auswirken. Die daraus resultierenden, kürzeren Iterationen zwischen Implementierung, Test und Korrektur sind damit einfacher zu überblicken und besser zu planen.

Auch wichtige Design-Verbesserungen, auf die normalerweise aus Risikogründen verzichtet werden muss, können mit dieser Vorgehensweise bewerkstelligt werden. Dadurch ist es möglich, das Design auf dem aktuellen Stand zu halten. Des Weiteren ist durch die Wahl kleinerer Iterationsschritte auch die Gefahr geringer, dass der Code außer Kontrolle gerät. Viele kleine Module, die jeweils einzeln getestet werden, sind überschaubarer und verlässlicher als ein großes Programm, das als Ganzes getestet werden muss.

Mit diesen Ansätzen beschäftigt sich die *agile Softwareentwicklung*, welche den iterativen Aspekt der Softwareentwicklung in den Vordergrund stellt und sich damit von der klassischen, geradlinigen Vorgehensweise (1. Plan, 2. Analyse, 3. Design, etc.) distanziert [SW07]. Das bekannteste, aus der agilen Softwareentwicklung hervorgegangene

Modell ist *Extreme Programming*, welches für die hier entwickelte Software allerdings nicht zum Einsatz gekommen ist und daher auch nicht ausführlicher beschrieben wird. Näheres kann u.a. in [Bec03], [SW07] und [Kel08] nachgelesen werden.

6.6.3 Auswirkungen des Frameworks auf das Design

Der Grundstein für ein gutes Design wird bereits mit der Wahl des Frameworks gelegt, welches einen großen Teil des Designs vorgibt. Hochwertige Frameworks nutzen dabei gezielt Mittel, um die Einhaltung des Designs sicherzustellen und unterbinden bzw. erschweren ungenaue, nicht der Konvention entsprechende Implementierungen.

Neben den Mitteln der objektorientierten Programmierung, über die z.B. Implementierungen erzwungen werden können, wird dabei auch von sog. *Asserts* Gebrauch gemacht. Diese prüfen im Entwicklungsstadium nicht definierte Zustände und Variablenbelegungen, und lösen gegebenenfalls einen sofortigen Abbruch mit ausführlicher Fehlermeldung aus. Auf diese Weise können frühzeitig interne Fehler und unzulässige Zustände, die nicht zwangsläufig unmittelbare Folgen haben, schon während der Entwicklung erkannt und ausgebessert werden. In der Regel können diese Prüfungen über *Precompiler*- oder andere Konfigurationseinstellungen für die Erstellung der finalen Version deaktiviert werden.

Ein Beispiel für ein vorbildliches *Assert* aus dem hier gewählten *JUCE*-Framework ist die Prüfung unsicherer Zugriffe auf die Benutzeroberfläche. Die *Assert*-Meldung ist dabei aussagekräftig formuliert, wodurch das Problem im Programm schnell gefunden werden kann.

6.6.4 Testbarkeit

Neben den Überlegungen, wie Problemen auf struktureller Ebene begegnet und eine objektorientierte Lösung gefunden werden kann, sollte auch die Testbarkeit im Design verankert sein. Dieser Aspekt wird auch häufig unter dem Begriff *Design For Testability* [Mes07] geführt.

Die zugrundeliegende Idee ist eine möglichst lose Bindung zwischen den Klassen und Objekten, wodurch das getrennte Testen einzelner Software-Teile erleichtert oder sogar erst ermöglicht wird. Dieses Ziel steht im Einklang mit den grundsätzlichen Design-Zielen, womit der dafür benötigte Aufwand gerechtfertigt werden kann.

Je weiter man sich von der agilen Softwareentwicklung entfernt, desto schwieriger wird die Realisierung einer umfangreichen Testbarkeit. Im Gegensatz zu extremen Ansätzen, wie z.B. der *testgetriebenen Entwicklung*, bei welcher zuerst der Test und erst anschließend die zugehörige Implementierung geschrieben wird, ist es in der klassischen Software-Enwicklung schwierig, von Anfang an alle notwendigen Tests vorherzusehen und das Design dahingehend zu optimieren. [Mes07]

Dem könnte man entgegnen, dass auch in der klassischen Softwareentwicklung neu hinzugefügte Code-Teile während der Entwicklungsphase getestet werden. Als Werkzeug kommt dabei ein sog. *Debugger* zum Einsatz. Dieser ermöglicht die schrittweise Ausführung des Programms, die Überwachung von Variablen, etc. Der größte Kritikpunkt dabei ist, dass die Bedienung nur schwer automatisierbar ist und somit mit einem hohen manuellen Aufwand verbunden ist. Dem setzen diese Werkzeuge zahlreiche Funktionen entgegen, die, wenn überhaupt, nur mit erheblichem Aufwand in Testprogrammen umsetzbar wären.

Ein ebenfalls oft genanntes Argument ist der Vorzug eines Tests gegenüber einer Fehlersuche. Auch diese Aussage kann entkräftet werden, da das Kernziel eines jeden Tests das Aufdecken von Fehlern ist [Mye04], und damit beim Test wie auch bei der Fehlersuche ein und dasselbe Ziel verfolgt wird. Folgendes Zitat von Dijkstra unterstreicht diese Aussage: „Program testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence.“ [Dij72]

Wird ein Mittelweg zwischen klassischer und agiler Softwareentwicklung verfolgt, wie es auch hier der Fall ist, so spricht nichts dagegen, die Möglichkeiten beider Welten zu kombinieren. Zu klar getrennten Teilen, wie z.B. im konkreten Fall der Transformation, können automatisierte Tests geschrieben und als Regressionstests laufend eingesetzt werden. Schwer zugängliche Stellen, wie z.B. fremde Klassen, können dagegen mittels Debugger auf Funktionstüchtigkeit und Integrationsfehler überprüft werden. Speziell, wenn spätere Änderungen an diesen Stellen unwahrscheinlich sind, ist die Anwendung eines Debuggers keinesfalls unvorteilhaft.

Schließlich sei noch darauf hingewiesen, dass auch der Einsatz der zuvor in Abschnitt 6.6.3 erwähnten *Asserts* erheblich zur Testbarkeit und Qualität des Codes beitragen kann. Hierbei werden in den Quell-Code Prüfungen unzulässiger Zustände etc. eingebaut, die im Entwicklungsmodus zur Laufzeit aktiv sind. Trifft eine dieser *Assert*-Bedingungen zu, dann wird das Programm sofort an dieser Stelle abgebrochen. Damit lassen sich interne Fehler sehr schnell aufdecken, die sich ansonsten nur indirekt auswirken würden und dadurch schwer zu finden wären.

6.6.5 Vorgehensweise anhand der Transformation

Wie bereits in Abschnitt 6.5 erwähnt, ist es zielführend, im Design eine Verallgemeinerung und damit eine Abstraktion der verschiedenen Transformations-Implementierungen vorzusehen. Am Beispiel dieses Vorhabens werden grundlegende Prinzipien und Vorgehensweisen, welche beim Design zum Einsatz kamen, ausführlich dargelegt. In den anschließenden Abschnitten werden die verbleibenden Design-Lösungen nur mehr grob vorgestellt, da diese nach dem gleichen Prinzip erarbeitet wurden.

Bei einem ersten Entwurf mit nur einer Transformationsart (z.B. FFT) ist der Bedarf einer Abstraktion noch nicht gegeben. Sobald aber weitere Transformationsarten hinzukommen, drängt sich die Erweiterung des Designs auf, da ansonsten vermehrt zu unvorteilhaften Mitteln wie globalen Schaltern oder unübersichtlich vielen Abzweigungen gegriffen werden müsste.

Grundsätzlich lässt sich der Design-Ansatz über folgende Überlegungen finden:

1. **Anbindung an die Audioverarbeitung**: Für die angrenzende Audioverarbeitung ist es nicht von Interesse, wie die Transformation im Detail funktioniert. Sie benötigt lediglich einen Abnehmer für die Audiodaten.
2. **Anbindung an die Benutzeroberfläche**: Für die angrenzende Benutzeroberfläche ist es ebenfalls nicht von Interesse, wie die internen Abläufe der verschiedenen Transformationen gestaltet sind. Es wird lediglich ein einheitlicher Zugang zu den Spektraldaten gefordert.
3. **Erzeugung der Transformation**: Der Benutzer kann über die Oberfläche unterschiedliche Transformationsarten auswählen. Je nach Wahl soll damit das entsprechende Transformationsobjekt erzeugt werden. Wie diese Erzeugung im Detail funktioniert und welche konkrete Transformations-Implementierung dafür zum Einsatz kommt, ist von außen gesehen nicht relevant. Daher soll auch nur eine vereinfachte, einheitliche Sicht der erzeugten Transformation zurückgeliefert werden, die das vom Umfeld gewünschte Verhalten zeigt und für alle Transformationsarten identisch ist. Dadurch müssen außenstehende Objekte nicht mehr zwischen den Transformationsarten unterscheiden. Umgekehrt kann auf einfache Weise eine neue Transformation implementiert werden, ohne dass der außenstehende Code angepasst werden muss.
4. **Verständigung über neu verfügbare Spektraldaten**: Eine Aufbereitung der Spektraldaten kann erst dann erfolgen, wenn neue, vollständig berechnete Daten verfügbar sind. Die Aufbereitung sollte aber für eine bessere Modularität in einer getrennten Klasse implementiert werden. Somit wäre es vorteilhaft, wenn die Aufbereitung ereignisorientiert über neue Spektraldaten verständigt werden könnte.

Abstrakte Klassen

Aus den vorangegangenen Überlegungen, insbesondere aus den Punkten 1 und 2, lässt sich eine zentrale Forderung ablesen: Es muss eine vereinfachte und einheitliche Sicht auf die verschiedenen Transformationen definiert werden. Dazu bedient man sich in der objektorientierten Programmierung einer sog. *abstrakten Klasse*. Diese kann im Gegensatz zu herkömmlichen Klassen nicht instanziiert werden, d.h. es können keine Objekte daraus erzeugt werden. Welchen Zweck erfüllt sie dann? Einsteigern wird oft erklärt, dass sich abstrakte Klassen gut als Ausgangspunkt größerer Vererbungsstrukturen einsetzen lassen, bei welchen eine Instanziierung der Ausgangsklasse nicht erwünscht ist. Hinter dieser vereinfachten Sichtweise verbirgt sich aber ein wesentlich größeres Potential: Abstrakte Klassen beinhalten abstrakte Methoden, die nicht implementiert werden und somit nur einen Rahmen vorgeben. Die ableitenden Klassen werden *gezwungen*, diese Methoden zu implementieren. Diese Eigenschaft verleiht abstrakten Klassen die Fähigkeit, Standards zu repräsentieren.

Wendet man diese Fähigkeit abstrakter Klassen am konkreten Beispiel der Transformation an, so kann man sagen, dass jede Transformation einen Mindeststandard erfüllen muss, welcher in der abstrakten Klasse mit dem Namen **Transformation** festgehalten wird. Darin muss mitunter eine Methode zum Einlesen der Abtastwerte und eine zum Auslesen der Spektraldaten vorhanden sein. Jede Klasse, die eine spezielle Form der Transformation implementieren möchte (z.B. FFT), muss diesen Standard erfüllen. Das geschieht, indem diese von der abstrakten Klasse abgeleitet wird. Somit wird sichergestellt, dass die in der abstrakten Klasse festgelegten Methoden exakt mit dem gleichen Namen und den gleichen Übergabeparametern in der abgeleiteten Klasse implementiert werden. Von außen betrachtet können damit alle Transformationen, die diesen Standard erfüllen, einheitlich angesprochen werden. Es sind zwar keine Besonderheiten einzelner Transformationen aufrufbar, diese spielen für die außenstehenden Klassen aber ohnehin keine Rolle.

Abbildung 6-2 zeigt im Wesentlichen das Design der Transformationsklassen, das die zuvor genannten vier Überlegungspunkte umsetzt. Man kann hieraus erkennen, dass die zentrale, abstrakte Klasse **Transformation** zum einen im herkömmlichen Sinne als Ausgangspunkt für Vererbung dient und somit allgemeine Vorgänge, wie z.B. den Ablauf der Berechnung, kapselt (siehe Methode **calculate**). Zum anderen dient die abstrakte Klasse **Transformation** aber auch als *Standard* und erlegt damit allen ableitenden Klassen auf, die definierten Standardmethoden, wie z.B. jene für die Ein- und Ausgabe, zu implementieren. Am Rande sei noch erwähnt, dass in anderen objektorientierten Programmiersprachen, wie z.B. *Java*, diese *Standards* in Form sog. *Interfaces* definiert werden und sich auf diese Weise syntaktisch deutlicher von herkömmlichen abstrakten Klassen absetzen.

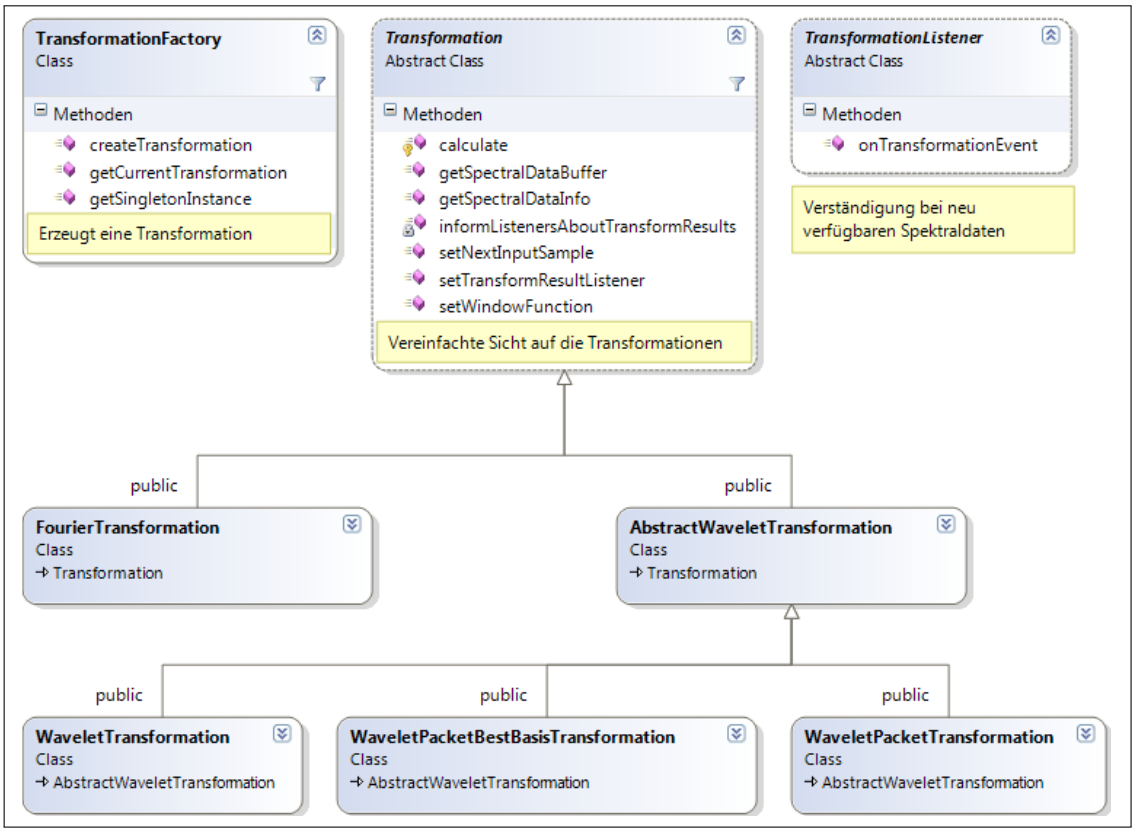

Abbildung 6-2: Klassendiagramm der Transformation

Fabrikmethode

In Abbildung 6-2 ist noch eine weitere Klasse dargestellt: Die **TransformationFactory**. Diese setzt die zuvor unter Punkt 3 genannte Überlegung bezüglich einer einfachen Erzeugung von Transformationen um. Die Aufgabe dieser Klasse ist es, das gewünschte, spezielle Transformationsobjekt zu erzeugen und alle damit verbundenen Schritte, wie Initialisierungen, Parametrisierungen, etc., abzuwickeln. Nach außen wird nur die vereinfachte Sicht, also die abstrakte Klasse **Transformation**, zurückgeliefert. Das entspricht einer Variante der sog. *Fabrikmethode* bzw. *Factorymethod* [GHJV10], welche zu den *erzeugenden Entwurfsmustern* nach *E.Gamma, R.Helm, R.Johnson, J.Vlissides* gehört.

Singleton

Zusätzlich wurde beim Design der Klasse **TransformationFactory** noch ein zweites Entwurfsmuster angewandt, das als *Singleton* bezeichnet wird. Mit diesem Entwurfsmuster kann erreicht werden, dass nur eine Instanz einer Klasse erzeugt wird und diese global angesprochen werden kann [GHJV10].

Auf diese Weise kann eine gänzlich statische Klasse und damit die Offenlegung aller zu verbergenden, internen Abläufe der Klasse verhindert werden. Des Weiteren sind statische Klassen nur in Zusammenhang mit globalen Konstanten oder sehr elementaren, unabhängigen Routinen sinnvoll. Sie verleiten schnell zur unvorteilhaften Programmierung, z.B. dem Einsatz globaler Variablen, wovon in jedem Fall abzuraten ist. Darüber hinaus sind diese in der Regel nicht *thread-safe*, gelten also als unsicher bei paralleler Verarbeitung.

Singletons sind häufig in Kombination mit erzeugenden Entwufsmustern, wie z.B. der *Fabrikmethode*, anzutreffen, da bei einer Erzeuger-Klasse selten mehr als eine Instanz benötigt wird.

Beobachter

Schließlich muss noch ein Weg gefunden werden, wie der zuvor genannte, vierte Überlegungspunkt umgesetzt werden kann, der eine ereignisorientierte Verständigung bei neu verfügbaren Spektraldaten vorsieht. Abbildung 6-2 zeigt die Lösung, welche mithilfe der Klasse **TransformationListener** umgesetzt wurde. Hierbei handelt es sich ebenfalls um ein Entwurfsmuster, das als *Beobachter* bzw. *Observer* bezeichnet wird [GHJV10]. Auch bei diesem Entwurfsmuster wird die Fähigkeit abstrakter Klassen genutzt, einen *Standard* zu repräsentieren, wobei in diesem Fall das Kapseln von Verhalten bezweckt wird.

Das Prinzip des *Beobachter-Musters* kann einfach am Beispiel eines Newsletters erklärt werden: Der Interessent gibt dem Versender seine E-Mail-Adresse bekannt und registriert sich damit für den Newsletter. Der Versender verschickt Neuigkeiten immer an alle in der Liste eingetragenen Interessenten. Dabei muss er darauf vertrauen, dass diese einen bestimmten Mindeststandard erfüllen, zu welchem z.B. eine gültige E-Mail-Adresse zählt. Ist dieser Mindeststandard erfüllt, bekommt der Interessent regelmäßig Informationen, bis er das Newsletter-Abonnement kündigt und seine E-Mail-Adresse aus der Liste des Versenders entfernt wird.

Im konkreten Fall der Transformation ist der Interessent die Benutzeroberfläche, die bei neu verfügbaren Spektraldaten verständigt werden soll. Der Versender ist die Transformation, welche nach erfolgreicher Berechnung eine Verständigung an alle eingetragenen Interessenten liefern soll. Damit das möglich ist, muss die Transformation wissen, welche Methode des Interessenten im Zuge der Verständigung aufzurufen ist. Dieses Problem wird mithilfe der abstrakten Klasse **TransformationListener** gelöst, die den vom Interessenten zu erfüllenden Standard vorgibt. Der Interessent, also hier die Benutzeroberfläche, muss u.a. eine Ableitung der **TransformationListener**-Klasse sein,

damit sie sich als Empfänger für die Verständigung mittels der Methode **setTransformResultListener** bei der Klasse **Transformation** registrieren kann. Zusätzlich muss die Benutzeroberfläche die Methode **onTransformationEvent** implementieren, worin all jene Aktionen implementiert werden, die bei neu verfügbaren Spektraldaten abgearbeitet werden müssen.

Anwendung von Entwurfsmustern

Die oben genannten Entwurfsmuster, aber auch andere, sollten als Ausgangsbasis und Ideensammlung angesehen werden, keinesfalls aber als Garant für ein gutes Design. Design hat nach wie vor viel mit kreativer Problemlösung zu tun. Vorgefertigte Musterlösungen können als Ergänzung einen wertvollen Beitrag leisten, sie liefern aber kein fertiges Design [Bud03]. Auch die oben genannten Entwurfsmuster wurden hier nicht direkt übernommen. Beispielsweise wurde keine Liste für Klassen vorgesehen, die über neu verfügbare Spektraldaten zu verständigen sind, sondern nur eine einfache Variable, da vorerst nur eine Klasse verständigt werden soll. Auch bei der *Fabrikmethode* wurde eine, wenngleich weit verbreitete, Variante gewählt.

Genauere Informationen bezüglich Software-Design bzw. Entwurfsmuster können u.a. in [Bud03] und [GHJV10] nachgelesen werden.

6.6.6 Fensterung

Das Design der Fensterung, welche insbesondere in Zusammenhang mit der FFT zu

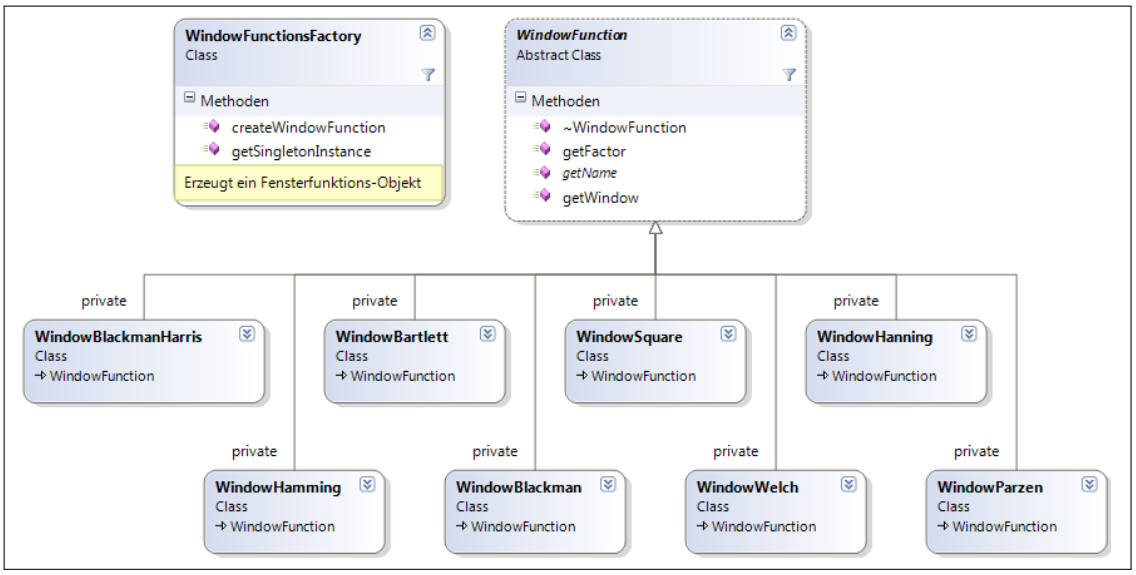

Abbildung 6-3: Klassendiagramm der Fensterung

einer besseren Spektraldarstellung eingesetzt wird (siehe Abschnitt 2.3), entspricht prinzipiell dem der Transformation. Auch hier wird eine Vereinheitlichung dazu verwendet, verschiedene Implementierungen nach außen hin zu verbergen und gleichzeitig eine implementierungsunabhängige Anbindung der Fensterfunktionen zu ermöglichen. Das zugehörige Klassendiagramm ist in Abbildung 6-3 dargestellt.

6.6.7 Plugin-Schnittstelle

Die Plugin-Schnittstelle bildet den zentralen Ausgangspunkt des Programms und wird weitgehendst vom *JUCE*-Framework abgedeckt. Es muss lediglich eine neue Klasse

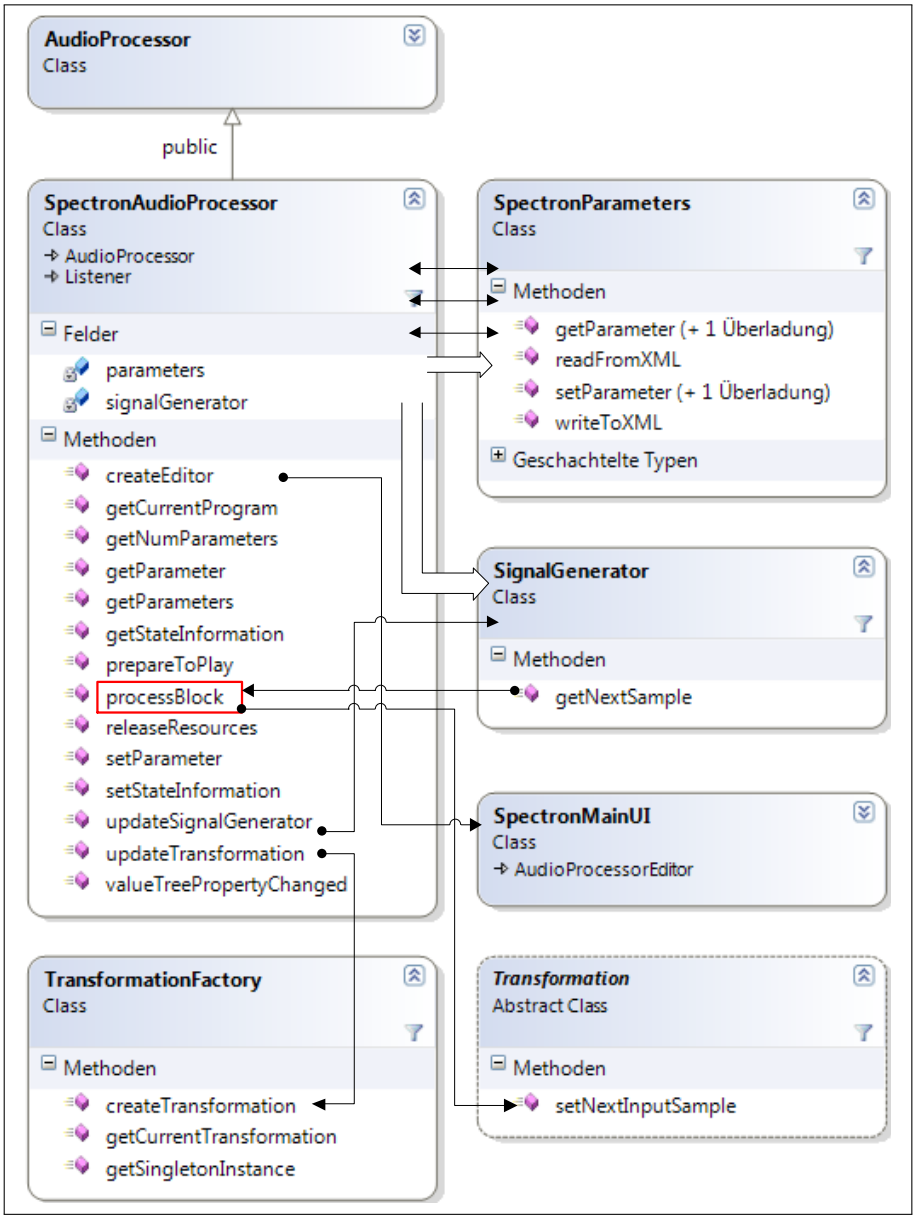

Abbildung 6-4: Klassendiagramm der Plugin-Schnittstelle

erstellt werden, welche sich von der Basisklasse **AudioProcessor** ableitet. Dort werden die individuellen Implementierungen durch Überschreiben der dafür vorgesehenen Methoden realisiert. Ein Beispiel dafür ist die Methode **processBlock**, in welcher die Audioverarbeitung implementiert wird.

Abbildung 6-4 zeigt das Klassendiagramm der Schnittstelle und das direkt damit zusammenhängende Umfeld. Alle hier nicht relevanten Methoden, Variablen, etc. wurden für eine bessere Übersicht weggelassen. Zudem sei darauf hingewiesen, dass zum Zeitpunkt der Entwicklung der Name des Plugins noch nicht feststand. Aus diesem Grund sind noch zahlreiche Klassen unter der bisherigen Bezeichnung *Spectron* zu finden, obwohl der Name zwischenzeitlich auf *Speclet* geändert wurde.

Die Handhabung der Plugin-Parameter, welche die Einstellungen repräsentieren, wurde im *JUCE*-Framework in Anlehnung an das VST-SDK designt und erweist sich dadurch als etwas umständlich. Zudem ist keine klare Zuständigkeitstrennung gegeben, denn die Klasse, welche die Audioverarbeitung abwickelt, ist gleichzeitig für die Parameterverwaltung zuständig. Aus diesem Grund wurde die Parameterverwaltung in die Klasse **SpectronParameter** ausgelagert. Diese wurde so designt, dass darin vorgenommene Erweiterungen möglichst geringe Auswirkungen auf umliegende Klassen haben. Um eine möglichst lose Bindung zu gewährleisten, wurde auch hier das *Beobachter*-Muster angewandt, womit abhängige Klassen automatisch bei Parameteränderungen verständigt werden.

6.6.8 Benutzeroberfläche

Die Benutzeroberfläche, insbesondere in Kombination mit dem generierten Code des Oberflächen-Werkzeugs *Jucer*, weist die am stärksten durch das Framework vorgegebene Umsetzung auf. In Abbildung 6-5 ist das zugehörige Klassendiagramm dargestellt. Auch hier wurden sämtliche Methoden, Variablen und Klassen ausgeblendet, die für die Benutzeroberfläche nicht relevant sind.

Die Aufbereitung des Spektrums wurde in die Hilfsklasse **RenderingHelper** ausgelagert, in welcher die Spektraldaten aus dem Buffer **SpectralDataBuffer** zusammen mit Informationen über das Spektrum aus **SpectralDataInfo** in darstellbare Bildpunkte mit entsprechendem Farbton umgewandelt werden.

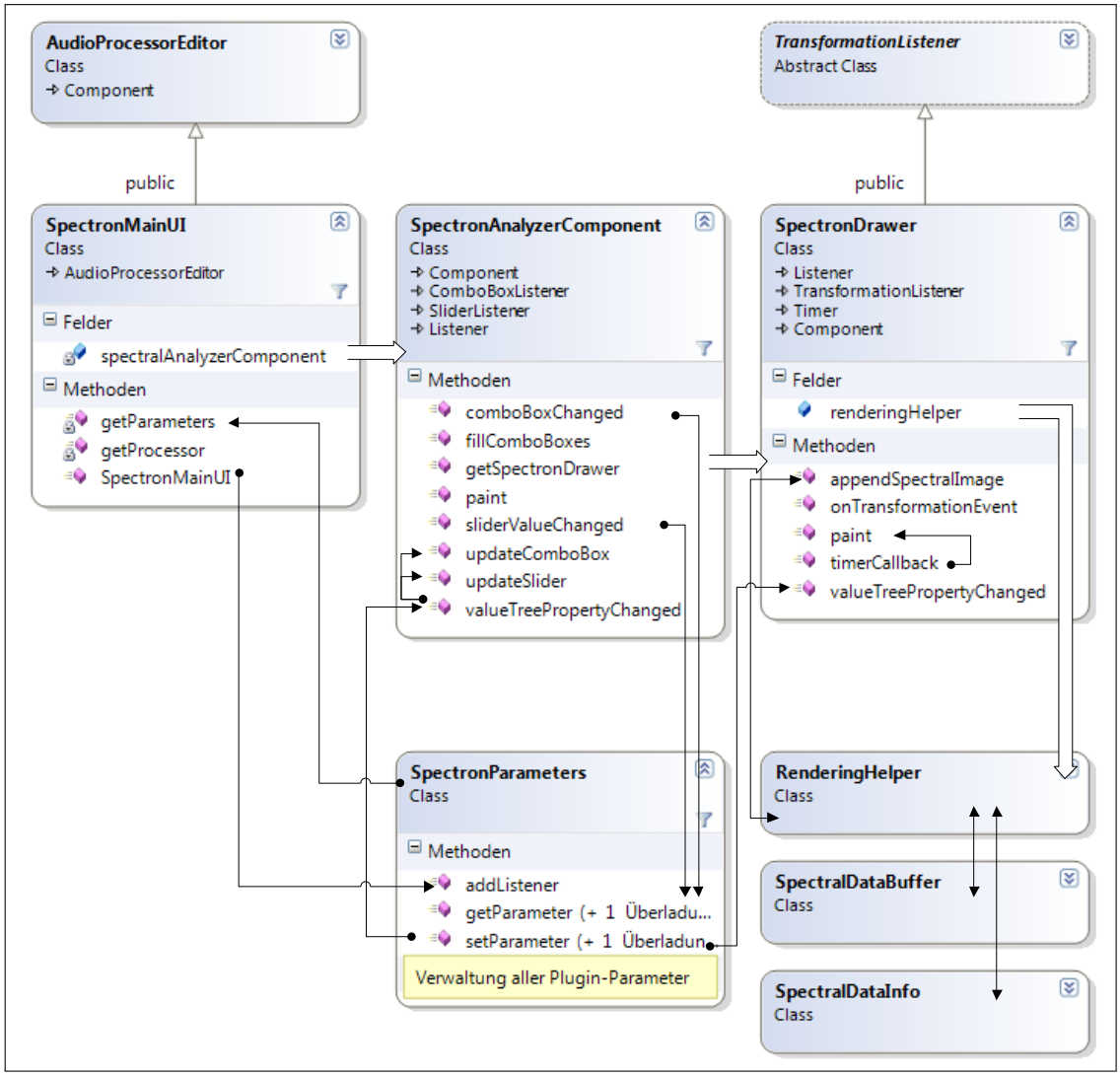

Abbildung 6-5: Klassendiagramm der Benutzeroberfläche

6.7 Software-Architektur

6.7.1 Einleitung

Im Zuge des Designs wurde bisher nur auf strukturelle, nicht aber auf die darunterliegenden, technischen Aspekte eingegangen. Die zuvor in Abschnitt 6.6.5 beschriebene, automatische Verständigung bei neu verfügbaren Spektraldaten ist beispielsweise eine rein strukturelle Lösung, welche nicht mit technischen Architekturprinzipien, wie betriebssystemnahen Ereignissen oder *Interrupts*, verwechselt werden darf.

In diesem Abschnitt wird auf einen Architekturaspekt detailliert eingegangen, welcher für das zu entwickelnde Plugin eine wichtige Rolle spielt: Das *Multithreading*.

6.7.2 Multithreading

Für eine bessere Auslastung der Ressourcen und des Prozessors, sowie geringere Wartezeiten für den Benutzer, unterstützen alle modernen Betriebssysteme parallel ablaufende Prozesse und darin wiederum parallel ablaufende *Threads*. Diese Unterstützung ist unabhängig von der Anzahl der tatsächlich im System installierten Prozessoren. Auch mit nur einem Prozessor können mehrere Prozesse und Threads *pseudo-parallel* laufen, indem diese vom Betriebssystem abwechselnd Zeitschlitze für die Verwendung des Prozessors zugeteilt bekommen [Tan09]. Eine besonders hohe Leistungssteigerung lässt sich unter Verwendung von Mehrkernprozessoren bzw. für Parallelverarbeitung vorgesehene Prozessorarchitekturen, wie z.B. Hyperthreading, erzielen [Wür08].

Während ein Prozess für das Betriebssystem einen hohen Verwaltungsaufwand bedeutet, sind Threads, die sich einen Prozess und die diesem zugeteilten Ressourcen teilen, wesentlich einfacher und damit schneller zu handhaben. Aus diesem Grund werden sie auch als *Leichtgewichtprozesse* bezeichnet. Jeder Prozess besteht aus mindestens einem Thread, in dem wiederum durch spezielle Betriebssystembefehle weitere hinzugefügt, modifiziert oder entfernt werden können. [Tan09]

Wann ist der Einsatz zusätzlicher Threads sinnvoll? Deren Einsatz ist speziell in Kombination mit jenen Programmteilen sinnvoll, welche möglichst unabhängig und parallel abgearbeitet werden können. Zudem sollten sie nicht nur CPU-intensive Berechnungen, sondern auch Betriebssystemzugriffe enthalten. Letztere führen zu Wartezeiten, während derer andere Threads, wie z.B. das Hauptprogramm, parallel weiterlaufen können. Nur in diesem Fall lassen sich mit Threads Leistungssteigerungen erzielen [Tan09].

Grundsätzlich sollte aber auch beim Einsatz von Threads folgender Leitsatz gelten: „So viele wie nötig, so wenige wie möglich.“ Die Programmierung von Multithreading-Anwendungen ist wesentlich aufwändiger, da der Code jederzeit vom Betriebssystem unterbrochen werden kann und somit Konflikte, z.B. beim parallelen Zugriff auf ein Speicherobjekt, entstehen können. Diese müssen berücksichtigt werden, da ansonsten die Stabilität des Programms gefährdet ist. Betriebssysteme stellen dazu sog. *Semaphore* bzw. *Mutexe* zur Verfügung, mit denen *kritische Abschnitte* gekennzeichnet werden können, die nur von einem bzw. einer begrenzten Anzahl von Threads gleichzeitig abgearbeitet werden dürfen. Näheres dazu kann in [Tan09] nachgelesen werden.

Prädestiniert für den Einsatz eines Threads ist die Benutzeroberfläche. Einerseits muss diese ständig mit Grafikroutinen des Betriebssystem interagieren, wodurch Wartezeiten entstehen. Andererseits erwartet der Benutzer eine möglichst rasche Reaktion auf die von ihm getätigten Aktionen, wie z.B. Tastatureingaben, ungeachtet jeglicher parallel ablaufender Berechnungen im Hintergrund. Das hier gewählte Framework *JUCE* unterstützt Multithreading und nutzt es auch intern zur Trennung von Hauptprogramm und Benutzeroberfläche. Von außen sind dazu keine weiteren Aktionen notwendig. Es bietet zudem Unterstützung bei der Behandlung von Konflikten, wie z.B. die Kennzeichnung kritischer Abschnitte. Diese Absicherung obliegt aber dem Entwickler.

7 Implementierung

7.1 Einleitung

Nach dem groben Überblick über Funktionen, Ziele und Nicht-Ziele in Kapitel 5 und der detaillierteren Beschreibung strukureller und technischer Vorgehensweisen in Kapitel 6, werden in diesem Kapitel die Details auf Code-Ebene erleutert. Dabei wird gezielt auf drei konkrete Problemlösungen und deren Implementierung eingegangen.

7.2 Sortierung der Wavelet-Paketknoten

Wird das Spektrum über die Wavelet-Pakettransformation (WPT) ermittelt, ist das Ergebnis nicht ohne weiteres grafisch darstellbar. Das Problem ist, dass die Ergebnisse zum einen in einer baumförmigen Struktur vorliegen und zum anderen nicht nach Frequenzen sortiert sind. Der erforderliche Sortieralgorithmus muss dabei unabhängig von der Baumstruktur sein, denn die Wahl der Knoten kann vom Benutzer oder automatisch beeinflusst werden. Näheres dazu kann in Abschnitt 2.4.9 nachgelesen werden. Das Ergebnis der WPT wird dort in Abbildung 2-16 veranschaulicht.

In Code-Ausschnitt 7.1 und 7.2 ist die Lösung des Problems dargestellt. Das Prinzip wurde von [Kap02] übernommen, welcher dieses in einem ausführlichen Beispiel darlegt.

```
//Sortiert die Baumstruktur nach absteigender Skala (aufsteigender Frequenz)
void sortWPTTreeByDescendingScale(const ArrayTreePer &tree) {
  if (!tree.origin) return;

  for (int level = 1; level <= (tree.maxlevel - 1); level++) {
    int blocksCount = tree.dim / tree.block_length(level);

    for (int block = 1; block < blocksCount; block+=2) {
    //Nur die ungeraden Knoten bzw. Bloecke, welche den Ausgaengen
    //der Wavelet-Hochpassfilter entsprechen, werden verarbeitet.
      swapWPTTreeChilds(tree, level, block);
    }
  }
}
```

Listing 7.1: Sortierung der Wavelet-Paketknoten - Hauptmethode

```
//Vertauscht das linke Element des
//Wavelet Paketknotens ("block") mit dem rechten
void swapWPTTreeChilds(
  const ArrayTreePer &tree,
  const integer &Level,
  const integer &Block)
{
  assert(Level >= 0);
  assert(Level < tree.maxlevel);
  assert(Block >= 0);
  assert(Block < (1<<Level));

  real_DWT* leftChild = tree.left_child(Level, Block);
  real_DWT* rightChild = tree.right_child(Level, Block);
  integer blocklength = tree.block_length(Level+1);

  for (int sublevel = 0; sublevel < tree.maxlevel - Level; sublevel++) {
  //Vertausche die Subelemente zum Erhalt der Konsistenz der Baumstruktur
    for (int element = 0; element < blocklength; element++) {
      int index  = element + sublevel*tree.dim;
      real_DWT helper = *(leftChild+index);
      *(leftChild+index)  = *(rightChild+index);
      *(rightChild+index) = helper;
    }
  }
}
```

Listing 7.2: Sortierung der Wavelet-Paketknoten - Hilfsmethode für Subelement

Das von der Bibliothek *wave++* vorgegebene, zu sortierende **ArrayTreePer**-Objekt, welches die Ergebnisse der WPT enthält, wird der Methode **sortWPTTreeByDescendingScale** als sog. *call-by-reference* Parameter übergeben. Durch die Übergabe der Referenz kann die Sortierung direkt auf das referenzierte Objekt durchgeführt werden, womit weder ein zusätzlicher Rückgabeparameter noch das Kopieren des Objekts notwendig ist.

Mittels Schleifen werden alle Ebenen, welche mit der Variable **level** indiziert sind, sowie alle darin enthaltenen Knoten, welche mit der Variable **block** indiziert sind, gelesen. Dabei werden nur die ungeradzahligen Knoten berücksichtigt, da diese die Ergebnisse der Hochpass-Filter bzw. die Wavelet-Koeffizienten enthalten.

Warum müssen speziell die Ausgänge der Hochpass-Filter genauer untersucht werden? Der elementare Unterschied zwischen der FWT und der WPT ist, dass bei letzterer nicht nur der Tiefpass-, sondern auch der Hochpass-Zweig einer weiteren Frequenzunterteilung unterzogen wird. Dabei tritt ein Phänomen auf, das durch die Abtastung und das

daraus resultierende, periodische Spektrum hervorgerufen wird: Auf jene Signale, die zuvor einen Hochpass-Filter zwecks Bandbreitenhalbierung durchlaufen haben, wirkt sich eine weitere Filterung spiegelverkehrt aus, da diese auf ein gespiegeltes Spektrum angewandt wird [JCH01]. Das bedeutet, dass eine weitere Hochpassfilterung den unteren Teilfrequenzbereich und eine weitere Tiefpassfilterung den oberen Teilfrequenzbereich liefert. Aus diesem Grund müssen alle unter dem Hochpasszweig liegenden Knoten vertauscht werden.

Das Vertauschen zweier Knoten wird durch die Hilfsmethode **swapWPTTreeChilds** (zu sehen in Code-Ausschnitt 7.2) durchgeführt. Diese muss zudem sicherstellen, dass die Konsistenz der Baumstruktur aufrecht erhalten bleibt, indem alle darunterliegenden Knoten ebenfalls vertauscht werden und somit der neuen Position des übergeordneten Knotens folgen.

Zu Beginn beider Methoden werden Eingabeparameter-Prüfungen durchgeführt. Warum werden Parameter von internen Methoden, die von außen nicht aufrufbar sind, geprüft? Während es als Selbstverständlichkeit gilt, externe Übergabeparameter zu prüfen, insbesondere wenn es sich um Eingaben von Benutzern handelt, wird von internen Übergabeparametern ein hohes Maß an Vertrauenswürdigkeit vorausgesetzt, ohne das zu prüfen. Das ist eine Lücke, die in der Praxis oft zu schwer auffindbaren Fehlern führt, da diese sich in der Regel nur indirekt auswirken und der Verdacht erst spät auf die ursächlichen Code-Stellen fällt. Absicherungen mit *asserts* eignen sich ideal zur vorsorglichen Absicherung solcher Fehler. Näheres dazu kann in Abschnitt 6.6.4 nachgelesen werden. Falls falsche Eingabeparameter unmittelbar ignoriert oder auf andere Weise behandelt werden können, eignet sich ebenfalls eine Platzierung dieser Befehle zu Beginn der Methode. Beispielsweise erfordert die Sortierung eines leeren Objekts keine Aktion, womit die Methode vorzeitig beendet werden kann.

7.3 Test der Wavelet-Transformation

Die folgenden Code-Ausschnitte 7.3 und 7.4 zeigen anhand eines Beispiels, wie die Tests implementiert wurden.

```
int _tmain(int argc, _TCHAR* argv[]) {
  TCommonSettings settings = setCommonSettings();
  //...
  assert(!test_WaveletTransormation(settings));
  //...
  return 0;
}
```

Listing 7.3: Teststeuerung

```
int test_WaveletTransormation (TCommonSettings settings) {
  //Erzeuge ein Wavelet-Transformations-Objekt
  Transformation* transformation =
  TransformationFactory::getSingletonInstance()
  ->createTransformation(
    SpectronParameters::TRANSFORM_FWT,
    settings.samplingRate,
    settings.resolution,
    SpectronParameters::WINDOWING_SQUARE,
    SpectronParameters::WAVELET_VAIDYANATHAN_18
  );
  assert(transformation);
  //...Hier ausgeblendet: Variablen-Deklarationen und Initialisierungen
  for (int t = 0; t < settings.resolution * 10; t++) {
  //Erzeuge Sinus-Testsignal + Transformation
    double sample = sin(omega * t * settings.samplingPeriod);
    transformation->setNextInputSample(sample);
  }
  //Lies Spektrum-Daten und -Informationen
  SpectralDataBuffer* buffer = transformation->getSpectralDataBuffer();
  SpectralDataInfo*   spectrumInfo = transformation->getSpectralDataInfo();
  //...Hier ausgeblendet: Diverse Pruefungen
  while (transformation->isOutputAvailable()) {
    //Lies das Spektrum des naechsten Zeitschlitzes
    SpectralDataBuffer::ItemType spectrum;
    buffer->read(&spectrum);
    //Lies das Spektrumstatistik
    SpectralDataBuffer::ItemStatisticsType statistics =
    buffer->getStatistics(&spectrum);
    //Ermittle die Spektrallinie mit der hoechsten Amplitude
    for (int i = 0; i < spectrum.size(); i++) {
      if (spectrum[i] == statistics.max) {
        indexOfMainSpectralLine = i;
        frequencyOfMax = i*(settings.samplingRate/settings.resolution/2.0f);
      }
    }
    //Pruefe, ob die Spektrallinie mit der hoechsten Amplitude
    //im Toleranzbereich rund um die Frequenz des generierten Testsignals
    //liegt. Das muss im Gegensatz zur FFT nicht immer der Fall sein und
    //wird daher statistisch gezaehlt.
    if (((frequencyOfMax + settings.frequencyResolutionInHz) >= frequency)
    &&  ((frequencyOfMax - settings.frequencyResolutionInHz) <= frequency)) {
      countOk++;
    } else {
      countWrongFrequency++;
    }
  }
  //...Hier ausgeblendet: Testresultat-Aufbereitung
}
```

Listing 7.4: Test der Wavelet-Transformation

Statt der einfachen Konsolenanwendung hätte man auch zu einem sog. *Unit-Test*-Framework greifen können, worauf hier aber aufgrund des Umfangs verzichtet wurde. In der Teststeuerung werden die einzelnen Testmethoden aufgerufen. Standardeinstellungen, wie z.B. die Abtastrate, werden über die Struktur **TCommonSettings** übermittelt.

In dem in 7.4 gezeigten Code-Ausschnitt wird ein Testsignal generiert, welches anschließend mit der Wavelet-Transformation analysiert wird. Die Prüfung des Spektrums erfolgt im letzten Schritt. Dort entscheidet sich, ob der Test erfolgreich war oder nicht. Da nur ein isolierter Teil der Software getestet wird, spricht man auch von einem *Unit-Test*, wenngleich das in 7.4 gezeigte Beispiel bezüglich Granularität, etc. eine eher pragmatische Lösung darstellt.

In Code-Ausschnitt 7.4 ist außerdem ein weiterer Vorteil von Tests dieser Art zu erkennen: Sie dienen auch als Dokumentation, da sie anhand eines Beispiels zeigen, wie die Verwendung der einzelnen Objekte und Klassen vorgesehen ist. In Code-Ausschnitt 7.4 ist beispielsweise zu erkennen, wie ein Wavelet-Transformations-Objekt erzeugt und parametrisiert wird.

7.4 Grafische Aufbereitung

Als letzte, detailliert ausgeführte Implementierung wird nun die Funktionsweise der grafischen Aufbereitung beschrieben. Wie in Abschnitt 6.6.8 erwähnt, befinden sich alle für die Aufbereitung notwendigen Hilfsmethoden in der eigens dafür vorgesehenen Klasse **RenderingHelper**. Die folgenden Code-Ausschnitte wurden aus dieser Klasse entnommen und zeigen die wesentlichen Bestandteile der Aufbereitung.

7.4.1 Hauptmethode

In Code-Ausschnitt 7.5 ist jene Methode dargestellt, welche von der Benutzeroberfläche zum Zeichnen des Spektrums aufgerufen wird. Dabei wird nur eine Zeiteinheit, also eine X-Position beachtet.

Damit der Zeichenvorgang nicht bei jedem Aktualisierungsvorgang der Benutzeroberfläche von neuem begonnen werden muss, wird das Spektrum in ein internes *Image*-Objekt geschrieben, welches schnell und einfach in der *paint*-Methode der Benutzeroberfläche am Bildschirm dargestellt werden kann. Zudem ist die Handhabung von Bildern einfacher, da für sie zahlreiche Funktionen, wie z.B. das *Scrollen*, zur Verfügung stehen.

```
//Hauptmethode zum Zeichnen der vertikalen Punkte des Spektrums
void RenderingHelper::renderVerticalPoints(
  Transformation* transformation,
  TAnalyzerSettings settings,
  long currentXPos,
  Image* spectralImage
){ //...Hier ausgeblendet: Eingabe-Pruefungen und Lesen des Spektrums
  int height  = spectralImage->getHeight();
  // Schleife ueber alle Pixel einer X-Position
  for (int i = 0; i <= height; i++) {
    //Ermitteln des Spektrallinienindex an der mitgegebenen Pixel-Position
    int spectralLineIndexOfPixel = pixelToIndex(i,height,info,isLogFreq);
    //Ermitteln der Amplitude der Spektrallinie
    double amplitude = spectrum[spectralLineIndexOfPixel];
    //Ermitteln der zugehoerigen Farbe
    double colorAmount = getColorAmount(amplitude,min,max,isLogAmpl);
    Colour colour = colorGradient.getColourAtPosition(colorAmount);
    //Zeichnen des Bildpunkts in das mitgegebene Image-Objekt
    spectralImage->setPixelAt(currentXPos,(height-i),colour);
  }
}
```

Listing 7.5: Hauptmethode zum Zeichnen der vertikalen Bildpunkte

7.4.2 Ermittlung des Spektrallinien-Index

Da der Ausgangspunkt der Verarbeitung die zu zeichnenden Pixel sind, muss ermittelt werden, welcher Spektralwert der jeweiligen Pixelposition zuzuordnen ist. Für eine bessere Übersicht wurde das in die Hilfsmethode **pixelToIndex** ausgelagert, welche im Code-Ausschnitt 7.6 zu sehen ist.

Zunächst wird die Y-Pixelposition in Relation zur Zeichenhöhe gesetzt, woraus sich die relative Y-Position in Form einer Gleitkommazahl zwischen Null und Eins ergibt. Neben Pixelposition und Zeichenhöhe werden dazu auch Informationen über das Spektrum, wie z.B. die minimal bzw. maximal darin enthaltene Frequenz, benötigt. Bei logarithmischer Frequenzachse wird nach dem gleichen Prinzip vorgegangen, allerdings muss hier die Berechnung über die Potenzfunktion erfolgen, welche die Umkehrfunktion des Logarithmus darstellt. Der Index lässt sich anschließend durch das Verhältnis der relativen Y-Position zur relativen Frequenzauflösung ermitteln. Die Gleichungen 7.1a und 7.1b zeigen die Umrechnung zwischen Pixelposition und Spektrallinien-Index in zusam-

mengefasster, mathematischer Form:

$$\Delta y_{lin} = \frac{y}{y_{min}}, \quad \Delta y_{log} = 10^{[(log(f_{max}) - log(f_{min})) \cdot \Delta y_{lin} + log(f_{min})]} \cdot \frac{1}{f_{max}} \tag{7.1a}$$

$$index_{lin} = \frac{\Delta y_{lin}}{\Delta f_{res}}, \quad index_{log} = \frac{\Delta y_{log}}{\Delta f_{res}} \tag{7.1b}$$

```
//Diese Methode ermittelt den Index der Spektrallinie, welche
//an der mitgegebenen Y-Pixelposition anzuzeigen ist
long RenderingHelper::pixelToIndex (
  int pixel,
  int height,
  SpectralDataInfo* info,
  bool logFrequency
) {
  //...Hier ausgeblendet: Eingabedaten-Pruefung(asserts,...)
  double frequencyResolution = info->getFrequencyResolution();
  double percentOfSpectrumPerIndex = info->getFrequencyPartitionSize();
  double percentOfSpectrum = pixel / (double)height;

  if (logFrequency) {
    double frequencyMax = info->getSamplingFrequency() / 2.0;
    double frequencyMaxLog = log10(frequencyMax);
    double frequencyMinLog = 1.0;
    double frequencyRangeLog = frequencyMaxLog - frequencyMinLog;
    percentOfSpectrum =
      pow(10, (frequencyRangeLog * percentOfSpectrum) + frequencyMinLog)
      / frequencyMax;
  }
  //...Hier ausgeblendet: Zwischenergebnis-Pruefung(asserts,...)
  int index = (int)ceil(percentOfSpectrum/percentOfSpectrumPerIndex);
  //Bereichsgrenzen pruefen und sicherstellen, Ergebnis zurueckliefern
  if (index < 0) index = 0;
  if (index >= frequencyResolution) index = (int)frequencyResolution - 1;
  return index;
}
```

Listing 7.6: Hilfsmethode zum Ermitteln des Spektrallinien-Index

7.4.3 Ermittlung des Farbtons

In Code-Ausschnitt 7.7 ist die letzte Hilfsmethode für die grafische Aufbereitung des Spektrums zu sehen, welche die Amplitudenwerte in Farbtöne umrechnet.

```
//Diese Methode berechnet den relativen Farbton zwischen 0.0 und 1.0
double RenderingHelper::getColorAmount(
  double amplitude,
  double minAmplitude,
  double maxAmplitude,
  bool logAmplitude
){
  //...Hier ausgeblendet: Eingabedaten-Pruefung(asserts,...)
  if (logAmplitude) {
  //Umwandlung in [dB], wenn gefordert
    maxAmplitude = 20 * log(maxAmplitude);
    minAmplitude = 20 * log(minAmplitude);
    amplitude = 20 * log(amplitude);
  }
  //Ermitteln des relativen Farbtons, welcher der Amplitude entspricht
  double rangeAmplitude = maxAmplitude - minAmplitude;
  double colorAmount = (amplitude - minAmplitude) / rangeAmplitude;
  //Bereichsgrenzen pruefen und sicherstellen, Ergebnis zurueckliefern
  if (colorAmount >= 1.0f) colorAmount = 1.0;
  if (colorAmount < 0.0)  colorAmount = 0.0;
  return colorAmount;
}
```

Listing 7.7: Hilfsmethode zur Berechnung des Farbwertes

Auch der Farbton wird in relativer Form, also als Gleitkommazahl zwischen Null und Eins, ausgegeben. Daraus kann dann, in Verbindung mit der *JUCE*-Framework-Klasse **ColourGradient**, die entsprechende Farbe direkt gewonnen werden.

Mit Hilfe der Statistikwerte, welche die minimale und die maximale Amplitude enthalten, kann mittels Schlussrechnung die relative Amplitude berechnet werden. Bei logarithmischer Darstellung werden die Amplitude und die Statistikwerte zuvor noch in Dezibel $[dB]$ umgerechnet.

Die Gleichungen 7.2a und 7.2b zeigen zusammenfassend die Umrechnung zwischen Pixelposition und Spektrallinien-Index in mathematischer Form:

$$\Delta color_{lin} = \frac{A - A_{min}}{A_{max} - A_{min}} \tag{7.2a}$$

$$\Delta color_{log} = 20 \cdot \frac{log(A) - log(A_{min})}{log(A_{max}) - log(A_{min})} \tag{7.2b}$$

8 Ergebnis

8.1 Einleitung

In diesem Kapitel wird das Ergebnis präsentiert und mit den Zielen aus Kapitel 5 verglichen. Anhand der abgebildeten Bildschirmausschnitte wird die Funktionsweise des Plugins im praktischen Einsatz demonstriert. Dabei wird insbesondere auf die vier Transformationsarten eingegangen.

8.2 Ergebnis der Entwicklung

Das unter dem Arbeitstitel *Speclet* umgesetzte VST-Plugin stellt das Ergebnis der Entwicklung dar. Neben dem Plugin, welches in Form einer *DLL*-Datei vorliegt, wurde zusätzlich eine sog. *Standalone-Version* in Form einer ausführbaren *EXE*-Datei erstellt. Damit ist es möglich, das Plugin als eigenständiges Programm aufzurufen, welches über einen grundlegenden Rahmen inklusive der damit verbundenen Audiotreiberanbindung verfügt. Für eine einfache Signalanalyse oder die Nutzung des internen Testsignalgenerators wird somit kein externes Programm benötigt. Auf diese Weise wurden auch die folgenden Bildschirmausschnitte erstellt.

Im ersten Bildschirmausschnitt 8-1 ist die Analyse eines sinusförmigen Testsignals un-

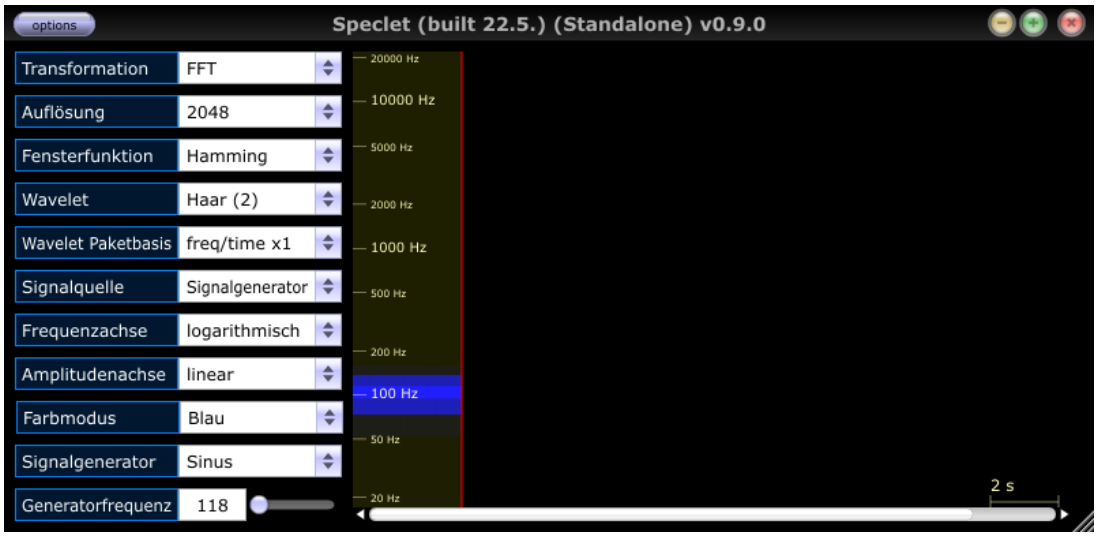

Abbildung 8-1: Speclet - Analyse eines Testsignals mittels FFT

ter Verwendung der FFT dargestellt. Die Parametrisierung des Plugins erfolgt über die Bedienelemente im linken Teil des Fensters. Dort können sämtliche Einstellungen, wie z.B. die Wahl der Transformationsart, etc., vorgenommen werden.

Aus den Einstellmöglichkeiten des Plugins lässt sich auch dessen Funktionsumfang ableiten. Die Parameter Auflösung, Signalquelle, Fensterfunktion, Amplitudenachse und Zeitachse bilden dabei die Grundfunktionen, welche zum Standard gehören und somit von einem Spektralanalyse-Werkzeug erwartet werden. Darüberhinaus sind auch spezielle Parameter, wie z.B. die Wahl der Transformation, enthalten, welche dieses Plugin von anderen unterscheidet. Vergleicht man die genannten Einstellungen und Funktionen mit den in Abschnitt 5.2 festgelegten Zielen, so kann nachgewiesen werden, dass alle Anforderungen erfüllt wurden.

In Abbildung 8-1 ist zudem zu erkennen, dass sich die FFT besonders für die Analyse sinusförmiger Signale eignet. Das Spektrum wird als durchgehende Linie dargestellt, welche je nach Auflösung und Fensterfunktion in Breite und Darstellungsschärfe variiert. Durch die logarithmische Frequenzachse wirkt die Spektrallinie bei tiefen Frequenzen breiter als bei hohen. Die FFT selbst weist aber eine lineare Frequenzauflösung auf, was bedeutet, dass die Breite jeder Spektrallinie unabhängig vom Frequenzbereich immer konstant ist.

Der folgende Bildschirmausschnitt 8-2 zeigt das Spektrum eines Sinussingals, welches mittels einer Wavelet-Transformation analysiert wurde.

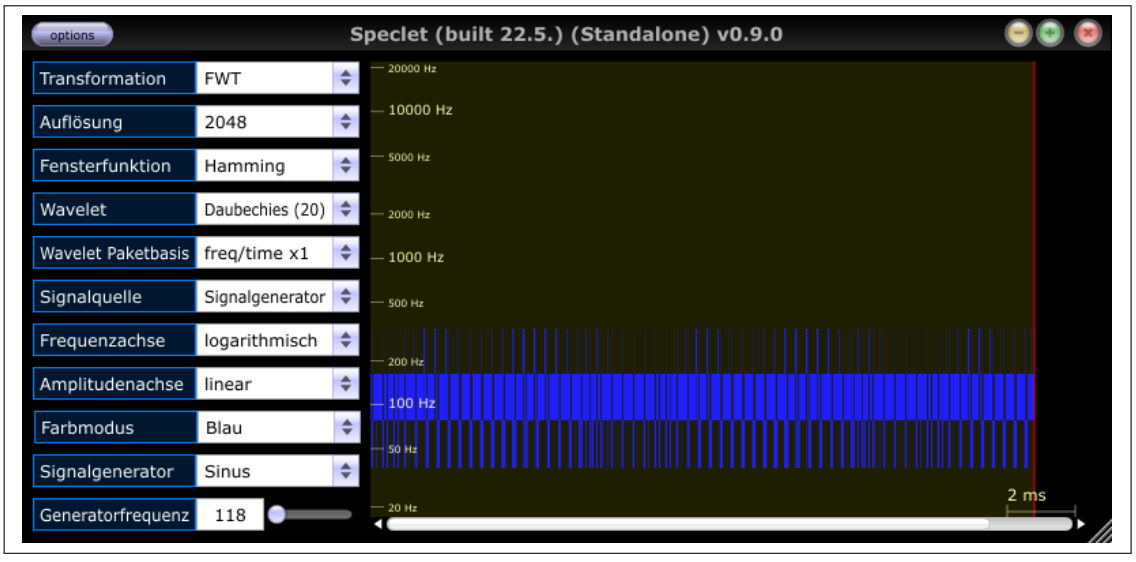

Abbildung 8-2: Speclet - Analyse eines Testsignals mittels FWT

Die FWT weist im tiefen Frequenzbereich eine der FFT vergleichbare Frequenzauflösung auf, welche aber zu den höheren Frequenzen hin abnimmt. Dies ist bei der logarithmischen Frequenzachse durch gleichbleibend breite Darstellung der Spektrallinien

erkennbar. Die Zeitauflösung verhält sich reziprok zur Frequenzauflösung, wodurch tiefe Frequenzen mit einer geringeren Zeitauflösung analysiert werden. Durch das Übersprechen benachbarter Frequenzbereiche scheint die Analyse des generierten Sinussignals zwar jener der FFT unterlegen zu sein, die FWT kann aber aufgrund ihrer Eigenschaften bei komplexen Signalen eine dem Gehör besser nachempfundene Darstellung des Spektrums erzielen.

Im nächsten Bildschirmausschnitt 8-3 ist die Analyse mittels Wavelet-Pakettransformation dargestellt, welche durch eine geeignete Wahl der Knoten einen Kompromiss zwischen Frequenz- und Zeitauflösung ermöglicht.

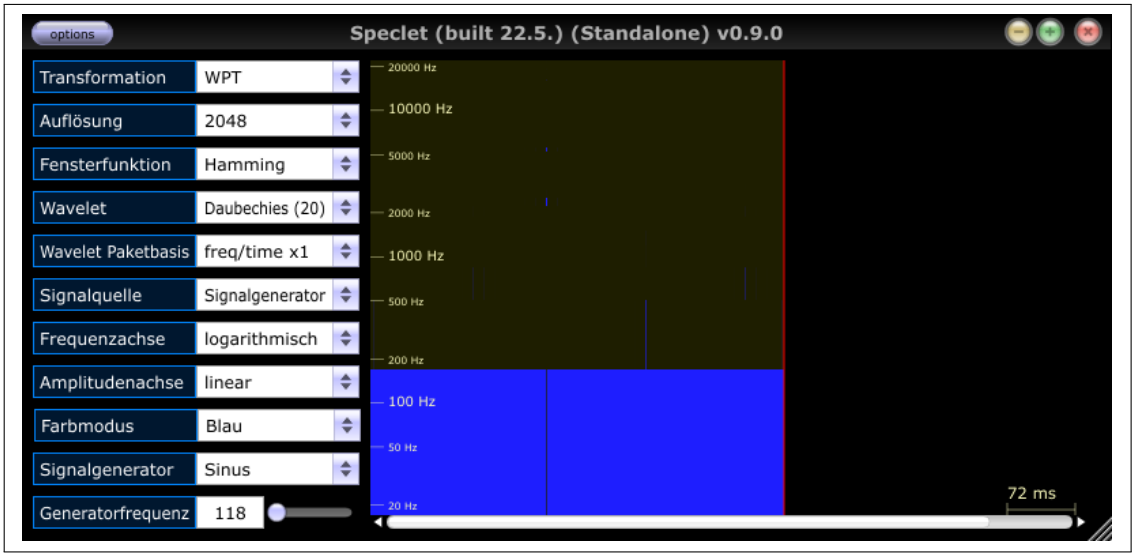

Abbildung 8-3: Speclet - Analyse eines Testsignals mittels WPT

Das Sinussignal wird zwar mit einer gegenüber der FFT und FWT deutlich schlechteren Frequenzauflösung dargestellt, das Übersprechen auf benachbarte Frequenzbereiche ist dagegen aber geringer. Des Weiteren geht mit der schlechteren Frequenzauflösung bei tiefen Frequenzen eine bessere Zeitauflösung einher. Die Knotenwahl kann durch die Wahl der Knotenebene mit dem Parameter *Wavelet Paketbasis* angepasst werden, wodurch sich die Frequenz- oder die Zeitauflösung jeweils um den Faktor 2 verfeinern lässt.

Der letzte Bildschirmausschnitt in Abbildung 8-4 zeigt eine erweiterte Form der WPT, bei der die Wahl der besten Knotenbasis durch einen Algorithmus erfolgt.

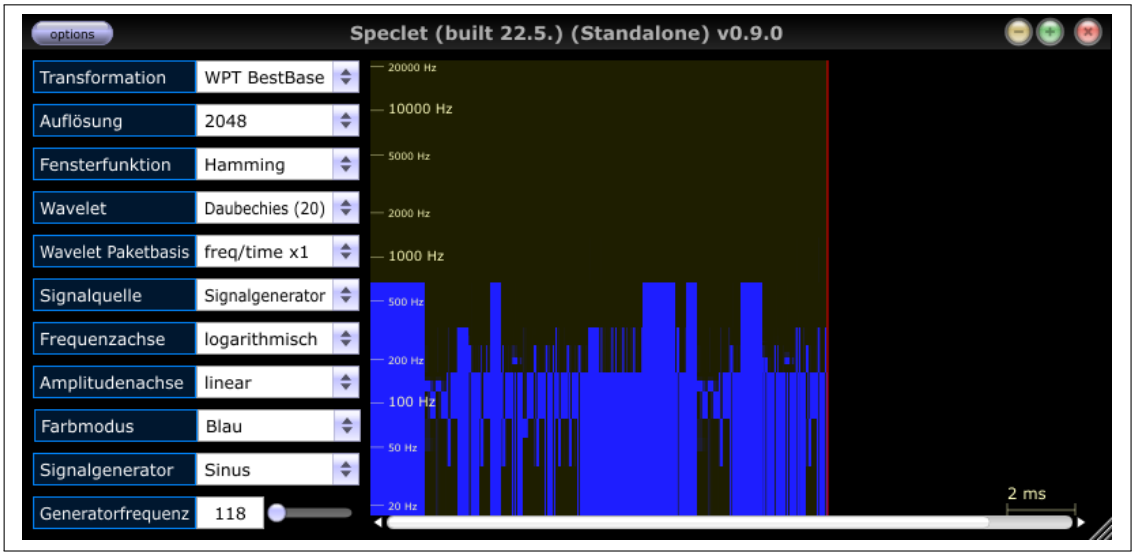

Abbildung 8-4: Speclet - Analyse eines Testsignals mittels WPT und bester Basis

Die bei jedem Analysevorgang neu errechnete Basis führt zu einem automatisch dem Signal angepassten Frequenz/Zeit-Raster. In Abbildung 8-4 ist das an der ständig wechselnden Frequenz- und Zeitauflösung zu erkennen, wodurch das Resultat dieser Transformationsform im konkreten Fall nicht überzeugen kann. Für stark variierende Signale birgt die adaptive Spektralanalyse aber ein großes Potential in sich, da sich die Zeit- und Frequenzauflösung laufend dem Signal anpasst und somit stets die aufschlussreichsten Details verfügbar sind.

9 Ausblick

9.1 Einleitung

In diesem abschließenden Kapitel wird das soeben präsentierte Ergebnis mit jenen Spektralanalyse-Plugins verglichen, welche in Kapitel 4 vorgestellt wurden. Aus den daraus gewonnen Erkenntnissen werden Ausbaumöglichkeiten der Funktionen, sowie der nicht-funktionalen Randbedingungen diskutiert.

9.2 Vergleich mit anderen Plugins

Im Vergleich zu den in Kapitel 4 vorgestellten Spektralanalyse-Plugins wirkt das entwickelte Plugin auf den ersten Blick etwas unausgereift. In puncto Bedienbarkeit, Benutzeroberfläche und Dokumentation wären noch einige Schritte nötig, um mit Plugins wie z.B. dem *Voxengo SPAN* oder dem *Waves PAZ* mithalten zu können. Der damit verbundene Aufwand konnte aber nicht in den Projektumfang aufgenommen werden, was bei der Festlegung der Nicht-Ziele in Abschnitt 5.2.4 genauer erläutert wurde.

Auch einige ergänzende Zusatzfunktionen, wie z.B. die Spektrumvergleichsfunktion des *Voxengo SPAN* oder die *RMS*-Funktion des *Waves PAZ*, konnten nicht innerhalb des Projektrahmens umgesetzt werden.

Die im Gegensatz zur klassischen Liniendarstellung gewählte Farbdiagramm-Darstellung ermöglicht dem Anwender im Vergleich zu anderen Spektralanalyse-Plugins die Analyse des zeitlichen Verlaufs des Spektrums, bietet aber ein weniger gewohntes Bild.

Von außen weniger auffallend, aber nicht minder wichtig, ist die interne Berechnung des Spektrums. In diesem Punkt ist das hier entwickelte Plugin gegenüber den Mitbewerbern im Vorteil. Die Auswahl verschiedener Analyseverfahren, insbesondere die adaptive Spektralanalyse, eröffnet zahlreiche neue Möglichkeiten. Aber auch auf diesem Gebiet sind Erweiterungen vorstellbar, speziell im Bereich alternativer Algorithmen zur Bestimmung der besten Basis.

9.3 Verfeinerungen und Erweiterungen

Dieser Abschnitt stellt auf den Funktionsumfang bezogene Verfeinerungen und Erweiterungen vor, welche als Ansatz für künftige Weiterentwicklungen dienen könnten. Die einzelnen Punkte werden dabei nur kurz und in aufzählender Form beschrieben.

Oberfläche und Bedienung

- **Hilfe und Dokumentation**: Neben einer Dokumentation könnte auch eine kontextbezogene Hilfe implementiert werden, welche gezielt zu Parametern oder anderen Oberflächenelementen Informationen bereitstellt. Die Umsetzung könnte in Form von sog. *Tooltips* oder einer kontextsensitiven Hilfe erfolgen. In diesem Zusammenhang könnten auch Ansätze von sog. *Teachware* einfließen, indem z.B. Abbildungen zu den einzelnen Wavelets und Fensterfunktionen zur Verfügung gestellt werden.
- **Maximierung der Spektraldarstellung**: Für eine effizientere Bildschirmflächennutzung wäre es vorteilhaft, wenn die Spektraldarstellung das gesamte Plugin-Fenster ausfüllt und die Einstellungen z.B. nur bei Bedarf eingeblendet werden.
- **Cursor und Zusatzinformation**: Viele Spektralanalyse-Plugins stellen in der Spektraldarstellung zusätzliche Mittel, wie z.B. einen Cursor zum Ablesen von Frequenzen und Spektralwerten, zur Verfügung. Erweiterungen dieser Art wären grundsätzlich mit geringem Aufwand verbunden, wobei hier die verschiedenen Transformationsarten zu berücksichtigen sind.
- **Timecode**: Wird das Plugin in eine Timecode-fähige Audio-Software eingebunden, könnte die spektrale Darstellung mit dem Timecode synchronisiert werden. Damit könnte man das Spektrum immer einem Takt bzw. einer genauen Position im Audiosignal zuordnen. Das ist besonders dann hilfreich, wenn in einer Schleife laufendes Audiomaterial untersucht werden soll.
- **Import, Export, Vergleich**: Das Importieren, Exportieren und Vergleichen von Spektren wäre ebenfalls eine sehr nützliche Funktionserweiterung, womit auch Änderungen im Spektrum untersucht werden könnten.

Spektraldarstellung

- **Zoom**: Sowohl die Zeit- als auch die Frequenzachse könnte von einer Zoom-Funktion profitieren, um Details sichtbar zu machen.

- **Farbverläufe**: Statt weniger, fest definierter Farbverläufe, wäre die Möglichkeit der Erstellung individueller Farbverläufe hilfreich. Dies hätte den Vorteil, dass eine größere Auswahl vordefinierter Farbverläufe mitgeliefert und individuell angepasst bzw. ergänzt werden könnte. Bei ähnlichen Spektralwerten besteht nämlich die Gefahr, dass deren Farbtöne nicht mehr unterscheidbar sind. Dem kann mit konfigurierbaren Farbverläufen gegengesteuert werden.
- **Amplitudennormalisierung**: Die Normalisierung der Amplitude wurde integriert, um sicherzustellen, dass stets die volle Bandbreite des definierten Farbverlaufs wirksam ist. Das führt aber bei Audiosignalen, die Pausen enthalten, zu wenig aussagekräftigen Spektralbildern. An Stelle der Pause wird kein leeres Spektrum, sondern ein mit sehr hoher Empfindlichkeit aus dem Rauschanteil ermitteltes Spektrum angezeigt. Um dieses Problem zu lösen, könnte eine Option zum Deaktivieren der Amplitudennormalisierung vorgesehen werden. Zudem könnte auch automatisch ein leeres Spektrum angezeigt werden, wenn die Spektraldaten einen unteren Amplitudengrenzwert unterschreiten.
- **Klassische Liniendarstellung**: Da viele Anwender die klassische Liniendarstellung von Spektralanalyse-Werkzeugen gewohnt sind und sich unter Umständen nicht mit der Farbdiagramm-Darstellung anfreunden können, wäre es möglich, diese ergänzend dazu als Darstellungsvariante zu realisieren.
- **Hybrid-Modus**: Durch die im Vergleich zur rein FFT-basierenden Spektralanalyse detailreicheren Zeitinformationen, stellt sich die Frage, wie sich diese zusätzlichen Daten am aussagekräftigsten darstellen lassen. Unter diesem Gesichtspunkt könnte eine Hybrid-Darstellung entworfen werden, welche das gewohnte Bild der Liniendarstellung um die Information des spektralen Verlaufs ergänzt und somit die Vorteile beider Darstellungen vereint.

Transformation

- **Ermittlung der besten Basis**: Beim Algorithmus zur Ermittlung der besten Basis für die WPT handelt es sich um einen, von der *wave++*-Bibliothek[17] vorgegebenen Algorithmus, welcher hauptsächlich für die Rauschunterdrückung und nicht für die Spektralanalyse vorgesehen ist. Die Entwicklung weiterer, auf die adaptive Spektralanalyse zugeschnittener Algorithmen wäre notwendig, um das volle Potential dieses Verfahrens zur Geltung zu bringen.
- **Knoten-Editor**: Eine ebenfalls interessante Erweiterung im Zusammenhang mit der WPT wäre ein Knoten-Editor, welcher in der Baumstruktur nicht nur die Auswahl der Knoten einer Ebene, sondern eine beliebige Zusammenstellung der Knoten ermöglichen würde. Damit ließen sich verschiedenste Frequenz/Zeit-Raster

[17] siehe Abschnitt 6.4.2

realisieren, wodurch neben der individuell konfigurierbaren Spektralanalyse auch eine sehr lehrreiche Veranschaulichung der WPT integriert werden könnte. Zudem könnte die Arbeitsweise des Algorithmus zur Ermittlung der besten Basis sichtbar gemacht werden.

- **Lifting**: Das *Lifting-Schema*, welches in Abschnitt 2.4.10 kurz beschrieben wurde, bietet eine interessante Alternative zur herkömmlichen, auf Filter basierenden Umsetzung der Wavelet-Transformation. Neben einer kürzeren Berechnungsdauer ließen sich damit auch auf Interpolation basierende Wavelets einsetzen.

Nicht-funktionale Randbedingungen

- **Kompatibilität**: Zusätzlich zur VST- und Standalone-Version bieten viele Hersteller ihre Plugins im *AU*-, *RTAS*, und *DirectX*-Format an, um einen größeren Kundenkreis anzusprechen. Auch die Erstellung einer 64-Bit-Version zählt mittlerweile zum Standard, da es zu Problemen mit 32-Bit Plugins kommen kann, wenn diese in 64-Bit Programme eingebunden werden.
- **Stabilität**: Um die Stabilität zu erhöhen, müssten noch weitere Tests, insbesondere in Bezug auf die parallele Abarbeitung von Threads, durchgeführt werden. Zudem müssten diese auf verschiedenen Plattformen erfolgen, um sicherzustellen, dass die Stabilität auch auf anderen Systemen gegeben ist.
- **Performance**: Um die Performance zu verbessern, wäre es notwendig, das Programm zu analysieren und alle problematischen Programmstellen zu optimieren. Als Performance-Analysewerkzeug wird dazu in der Regel ein sog. *Profiler* eingesetzt. Damit könnten zum einen der CPU-Verbrauch und zum anderen die Antwortzeiten reduziert werden. Häufig können im Zuge dessen auch interne Strukturfehler aufgedeckt werden, wodurch sich gleichzeitig die Qualität und Stabilität der Software steigern lässt.
- **Unit-Tests**: Um die Voraussetzungen für Stabilität, Performance und Erweiterbarkeit zu verbessern, wäre die Integration eines Unit-Testframeworks, sowie die Verfeinerung der Tests vorteilhaft. Dadurch könnten nicht nur neue Programmteile, sondern auch bestehende getestet werden. Wiederholte Tests bestehender Programmteile werden als *Regressionstests* bezeichnet und bilden die Grundlage für eine gute Erweiterbarkeit, insbesondere betreffend Designänderungen.

10 Schlusswort

Das Beschreiten neuer Wege in der Spektralanalyse stellt eine besondere Herausforderung dar. Bereits bewährte Verfahren, wie z.B. die FFT oder eine mittels Filterbank realisierte Terzbandanalyse, werden in zahlreichen Spektralanalyse-Plugins eingesetzt und liefern dabei sehr gute, ausgereifte Ergebnisse.

Die auf diesem Gebiet noch als exotisch geltenden Analyseformen werden nicht durch ihr großes Potential, sondern durch ihre Schwächen gegenüber herkömmlicher Verfahren wahrgenommen. Erst wenn es gelingt, diese Schwächen zu relativieren und deren Ausprägung durch geeignete Mittel, wie z.B. auf sie zugeschnittene Darstellungsformen, zu reduzieren, kann die Akzeptanz erhöht werden. Der Umstieg kann dabei durch Integration bewährter Verfahren, bekannter Paramter und gewohnter Darstellungen erleichtert werden.

Parallel dazu ist auch die Analyse mittels Wavelets und Wavelet-Paketen in der Praxis, speziell in Verbindung mit Spektralanalyse-Plugins, noch nicht vollständig ausgereift. Insbesondere die adaptive, also dem Signal angepasste, Spektralanalyse, welche mittels WPT und speziellen Algorithmen zur Ermittlung der besten Basis möglich ist, stellt aufgrund ihrer hohen Komplexität eine besondere Herausforderung dar.

Das entwickelte Plugin zeigt sowohl das große Potential und die damit verbundenen, neuen Möglichkeiten, als auch die noch zu bewältigenden Probleme und Herausforderungen. Es entspricht den im Projekt gesetzten Zielen und Anforderungen, lässt aber auch gleichzeitg die ausgegrenzten Punkte, wie z.B. die Marktreife, vermissen.

Trotz der zahlreichen Erweiterungs- und Verbesserungsideen, welche in dem vordefinierten Projektrahmen nicht Platz gefunden haben, kann das Ergebnis als Erfolg angesehen werden. Die daraus gewonnenen Erkenntnisse legen einen Grundstein für den Einsatz alternativer Spektralanalyseverfahren im Bereich der Audio-Plugins und stellen eine fundierte Grundlage für weitere Entwicklungen auf diesem Gebiet dar.

Literaturverzeichnis

[Add02] ADDISON, P.S.: *The illustrated wavelet transform handbook: introductory theory and applications in science, engineering, medicine and finance*. Institute of Physics Publishing, 2002. – ISBN 9780750306928

[Ant05] ANTONIOU, A.: *Digital signal processing: signals, systems and filters*. McGraw-Hill, 2005. – ISBN 9780071454247

[BA86] BERANEK, L.L. ; AMERICA, Acoustical S.: *Acoustics*. Published by the American Institute of Physics for the Acoustical Society of America, 1986 (Electrical and electronic engineering). – ISBN 9780883184943

[Bec03] BECK, K.: *Extreme Programming*. Addison-Wesley, 2003. – ISBN 9783827321398

[Bla03] BLATTER, C.: *Wavelets - eine Einführung*. Vieweg, 2003 (Advanced lectures in mathematics). – ISBN 9783528169473

[Bän05] BÄNI, W.: *Wavelets: Eine Einführung für Ingenieure*. 2. Oldenbourg, 2005. – ISBN 9783486577068

[Bud03] BUDGEN, D.: *Software design*. Addison-Wesley, 2003 (International computer science series). – ISBN 9780201722192

[Cha04] CHAU, F.: *Chemometrics: from basics to wavelet transform*. J. Wiley, 2004 (Chemical analysis Bd. 164). – ISBN 9780471202424

[Dau92] DAUBECHIES, Ingrid: *Ten lectures on wavelets*. 2. Society for Industrial and Applied Mathematics, 1992. – ISBN 9780898712742

[DGQ08] DECHEVSKY, Lubomir T. ; GRIP, Niklas ; QUAK, Ewald: *A comparative study of current Matlab and C++ wavelet software*. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.129.4835&rep=rep1&type=pdf. Version: 2008

[Dic87] DICKREITER, M.: *Handbuch der Tonstudiotechnik*. 5. Saur, 1987 (Bd. 1)

[Dij72] DIJKSTRA, Edsger W.: ACM Turing award lectures. Version: 1972. http://dx.doi.org/http://doi.acm.org/10.1145/1283920.1283927. New York, NY, USA : ACM, 1972. – DOI http://doi.acm.org/10.1145/1283920.1283927, Kapitel The humble programmer, 860–866

[DS98] DAUBECHIES, Ingrid ; SWELDENS, Wim: Factoring wavelet transforms into lifting steps. In: *J. Fourier Anal. Appl* 4 (1998), 247–269. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.106.8124

[Els00] ELSNER, N.: *Das Gehirn und sein Geist*. Wallstein, 2000. – ISBN 9783892444213

[EP09] EVEREST, F.A. ; POHLMANN, K.C.: *Master Handbook of Acoustics*. McGraw-Hill, 2009. – ISBN 9780071603324

[FJ] FRIGO, Matteo ; JOHNSON, Steven G.: *FFTW*. http://www.fftw.org, Abruf: 24.April 2011

[FJ98] FRIGO, Matteo ; JOHNSON, Steven G.: FFTW: An Adaptive Software Architecture For The FFT. In: *Proceedings of the 1998 IEEE International Conference on Acoustics, Speech and Signal Processing*, IEEE, 1998, 1381–1384

[FK01] FERRANDO, S.E. ; KOLASA, L.A.: Averages of best wavelet basis estimates for denoising. In: *Journal of Computational and Applied Mathematics* 136 (2001), Nr. 1-2, 357 - 367. http://dx.doi.org/DOI:10.1016/S0377-0427(00)00626-9. – DOI DOI: 10.1016/S0377–0427(00)00626–9. – ISSN 0377–0427

[Fla03] FLANAGAN, D.: *Java in a nutshell: deutsche Ausgabe fuer Java 1.4*. O'Reilly, 2003. – ISBN 9783897213326

[FZ07] FASTL, H. ; ZWICKER, E.: *Psychoacoustics: facts and models*. Springer, 2007 (Springer series in information sciences). – ISBN 9783540231592

[GHJV10] GAMMA, E. ; HELM, R. ; JOHNSON, R. ; VLISSIDES, J.: *Entwurfsmuster: Elemente wiederverwendbarer objektorientierter Software*. Addison Wesley

Verlag, 2010 (Programmer's Choice). – ISBN 9783827330437

[Grü08] GRÜNIGEN, D.C.: *Digitale Signalverarbeitung: Mit einer Einführung in die kontinuierlichen Signale und Systeme*. 4. Fachbuchverlag Leipzig, 2008. – ISBN 9783446414631

[Hof97] HOFFMANN, R.: *Signalanalyse und -erkennung*. Springer, 1997. – ISBN 9783540634430

[JCH01] JENSEN, A. ; COUR-HARBO, A.L.: *Ripples in mathematics: the discrete wavelet transform*. Springer, 2001. – ISBN 9783540416623

[Kap02] KAPLAN, I.: *Frequency Analysis Using the Wavelet Packet Transform*. Version: 2002. http://www.bearcave.com/misl/misl_tech/wavelets/packfreq/index.html, Abruf: 26.März 2011.

[Kel08] KELLY, A.: *Changing software development: learning to be agile*. John Wiley, 2008 (Safari Books Online). – ISBN 9780470515044

[KK06] KAMMEYER, K.D. ; KROSCHEL, K.: *Digitale Signalverarbeitung: Filterung und Spektralanalyse mit MATLAB-Übungen*. Teubner, 2006. – ISBN 9783835100725

[KWW] KÖBERL, Stefan ; WIMMESBERGER, David ; WINTER, Matthias: *Wavelets*. http://www.univie.ac.at/nuhag-php/login/skripten/data/Wavelets.pdf, Abruf: 16.April 2011

[LMR98] LOUIS, A.K. ; MAASS, P. ; RIEDER, A.: *Wavelets: Theorie und Anwendungen*. Teubner, 1998 (Teubner Studienbücher: Mathematik). – ISBN 9783519120940

[Mal99] MALLAT, S.G.: *A wavelet tour of signal processing*. 2. Academic Press, 1999 (Wavelet Analysis and Its Applications Series). – ISBN 9780124666061

[Mer99] MERTINS, A.: *Signal analysis: wavelets, filter banks, time-frequency transforms, and applications*. J. Wiley, 1999. – ISBN 9780471986263

[Mes07] MESZAROS, G.: *xUnit test patterns: refactoring test code*. Addison-Wesley, 2007 (The Addison-Wesley signature series). – ISBN 9780131495050

[Mey08] MEYER, M.: *Signalverarbeitung: Analoge und digitale Signale, Systeme und Filter*. Vieweg+Teubner Verlag, 2008 (Informations- und Kommunikationstechnik). – ISBN 9783834804945

[Moo07] MOORE, B.C.J.: *Cochlear hearing loss: physiological, psychological and technical issues*. John Wiley & Sons, 2007 (Wiley series in human communication science). – ISBN 9780470516331

[MW99] MADISETTI, V.K. ; WILLIAMS, D.B.: *Digital Signal Processing Handbook: Crcnetbase 1999*. Taylor and Francis, 1999. – ISBN 9780849321351

[Mye04] MYERS, Glenford J.: *The Art of Software Testing*. 2. John Wiley & Sons, 2004 (ISBN 0-471-46912-2)

[Pap62] PAPOULIS, A.: *The Fourier integral and its applications*. McGraw-Hill, 1962 (McGraw-Hill electronic sciences series)

[Pap09] PAPULA, L.: *Mathematik für Ingenieure und Naturwissenschaftler 1*. Vieweg+Teubner Verlag, 2009 (Viewegs Fachbücher der Technik Bd. 1). – ISBN 9783834805454

[Par10] PARK, T.H.: *Introduction to digital signal processing: computer musically speaking*. World Scientific, 2010. – ISBN 9789812790279

[Pas01] PASETTI, A.: *Software frameworks and embedded control systems*. Springer, 2001 (Lecture notes in computer science). – ISBN 9783540431893

[PM96] PROAKIS, J.G. ; MANOLAKIS, D.G.: *Digital signal processing: principles, algorithms, and applications*. Prentice Hall, 1996 (Prentice-Hall International editions). – ISBN 9780133737622

[PM08] POLYANIN, A.D. ; MANZHIROV, A.V.: *Handbook of integral equations*. 2. Chapman & Hall/CRC, 2008 (Handbooks of mathematical equations). – ISBN 9781584885078

[Rus92] RUSKAI, M.B.: *Wavelets and their applications*. Jones and Barlett, 1992 (Jones and Bartlett books in mathematics). – ISBN 9780867202250

[Sch05] SCHEITHAUER, R.: *Signale und Systeme*. Teubner, 2005. – ISBN

9783519164258

[Sen05] SENGPIEL, E.: *Normal equal-loudness-level contours - ISO 226:2003 Acoustics*. Version: 2005. http://www.sengpielaudio.com/Acoustics226-2003.pdf, Abruf: 27.März 2011

[Ste] STEINBERG MEDIA TECHNOLOGIES GMBH: *Steinberg Media Technologies*. http://www.steinberg.net/de/company/aboutsteinberg.html, Abruf: 10.April 2011

[Sun] SUN MICROSYSTEMS: *The Java™ Language: An Overview*. http://java.sun.com/docs/overviews/java/java-overview-1.html, Abruf: 18.April 2011

[SW07] SHORE, J. ; WARDEN, S.: *The art of agile development*. O'Reilly, 2007 (Theory in practice). – ISBN 9780596527679

[Swe97] SWELDENS, Wim: *The Lifting Scheme: A Construction Of Second Generation Wavelets*. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.50.7313. Version: 1997

[Tan09] TANENBAUM, A.S.: *Modern operating systems*. Pearson/Prentice Hall, 2009. – ISBN 9780138134594

[Val99] VALENS, C.: *The Fast Lifting Wavelet Transform*. http://polyvalens.pagesperso-orange.fr/clemens/download/tflwt.pdf. Version: 1999

[Val04] VALENS, C.: *A Really Friendly Guide to Wavelets*. Version: 2004. http://polyvalens.pagesperso-orange.fr/clemens/wavelets/wavelets.html, Abruf: 19.März 2011

[Van] VANEEV, Aleksey: *Voxengo SPAN - User Manual*. http://www.voxengo.com/product/span, Abruf: 2.April 2011

[Wav] WAVES: *PAZ Psychoacoustic Analyzer - User Manual*. http://www.waves.com/manuals/plugins/PAZ.pdf, Abruf: 1.April 2011

[Wür08] WÜRST, K.: *Mikroprozessortechnik: Grundlagen, Architekturen und Programmierung von Mikroprozessoren, Mikrocontrollern und Signalprozesso-*

ren. Vieweg+Teubner Verlag, 2008. – ISBN 9783834804617

[Zöb08] ZÖBEL, D.: *Echtzeitsysteme: Grundlagen der Planung*. Springer, 2008. – ISBN 9783540763956

Stichwortverzeichnis

Glossar

Header-Dateien	C- bzw. C++-sprachtypische Daten, in denen Deklarationen, wie z.B. von Klassen, vorgenommen werden
K-Metering	Von Mastering-Ingenieur Bob Katz definierte Metering-Standards
Mastering	Finale Bearbeitung von Audiomaterial und Vorbereitung zur Tonträgerherstellung
Mathcad™	Mathematik-Software von Parametric Technology Corporation (PTC)
Matlab™	Mathematik- und Ingenieur-Software von Mathworks Inc.
Metering	Anzeige der Lautheit- bzw. Lautstärke von Audiosignalen
Open Source	Software, deren Quellcode offengelegt ist
Precompiler	Vor der Kompilierung ablaufendes Übersetzungsprogramm, das u.a. durch Makros vertretene Code-Teile einsetzt
Rosa Rauschen	Rauschen, dessen Intensität mit steigender Frequenz um 3dB pro Oktave abnimmt und vom menschlichen Gehör als natürlich und ausgewogen empfunden wird
Timecode	Zusätzliche Zeitinformation aus dem Bereich der Audio- und Videobearbeitung, welche die exakte Position innerhalb eines Audio- bzw. Videoprojektes widerspiegelt und daher u.a. zur Synchronisation von Bild und Ton eingesetzt wird
White-Box-Tests	Tests, bei denen die von außen unsichtbaren, internen Abläufe der Software auf Fehler untersucht werden

Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und nur unter Verwendung der angegebenen Literatur und Hilfsmittel angefertigt habe.

Stellen, die wörtlich oder sinngemäß aus Quellen entnommen wurden, sind als solche kenntlich gemacht.

Diese Arbeit wurde in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt.

Mittweida, 2011